

大话存储

网络存储系统原理精解与最佳实践

张冬 编著

- ★ 国内首次全面披露网络存储深层技术细节
- ★ 特立独行的行文风格，一针见血的诠释技术
- ★ 网络存储和存储网络，谈笑间难点灰飞烟灭
- ★ 全面解密SAN、NAS系统
- ★ 涉及众多底层细节，中文资料独家提供
- ★ 提供最佳操作实践，严禁纸上谈兵
- ★ 四大网站本书讨论专题，你不是一个人在战斗

清华大学出版社

大话存储——网络存储系统原理精解与最佳实践

张　冬　编著

清华大学出版社
北　京

内 容 简 介

网络存储，是近二十年来的新兴行业。从纸带到硬盘再到大型磁盘阵列，存储系统经历了从简单到复杂，从单块硬盘到存储区域网络(SAN)。网络存储行业目前已经是一个步入正轨的IT行业了。

网络存储是一个涉及计算机硬件以及网络协议/技术、操作系统以及专业软件等各方面综合知识的领域。目前国内阐述网络存储的书籍少之又少，大部分是国外作品，对存储系统底层细节的描述不够深入，加之术语太多，初学者很难真正理解网络存储的精髓。

本书以特立独行的行文风格向读者阐述了整个网络存储系统。从硬盘到应用程序，这条路径上的每个节点，作者都进行了阐述。书中内容涉及：计算机 IO 基本概念，硬盘物理结构、盘片数据结构和工作原理，七种常见 RAID 原理详析以及性能细节对比，虚拟磁盘、卷和文件系统原理，磁盘阵列系统，OSI 模型，FC 协议，众多磁盘阵列架构，SAN 和 NAS 系统，TCP 和以太网以及 IP SAN，协议融合理论，存储虚拟化，存储及服务器群集，数据保护和备份技术，快照技术，数据容灾技术。

本书用独特的写作方式通俗地诠释了这些晦涩、枯燥的难点技术并提供了许多前所未有的操作实践和本书作者长期从事存储工作的一些经验点滴。

本书适合初入存储行业的技术工程师、售前工程师和销售人员阅读，同时适合资深存储行业人士用以提高技能，另外，网络工程师、网管、服务器软硬件开发与销售人员、Web 开发者、数据库开发者以及相关专业师生等也非常适合阅读本书。

图书在版编目(CIP)数据

大话存储——网络存储系统原理精解与最佳实践/张冬编著. —北京：清华大学出版社，2008.11
ISBN 978-7-302-18672-4

Ⅰ. 大…　Ⅱ. 张…　Ⅲ. 计算机网络—信息存储—研究　Ⅳ. TP393.0

中国版本图书馆 CIP 数据核字(2008)第 150498 号

责任编辑：栾大成
封面设计：杨玉兰
版式设计：北京东方人华科技有限公司
责任校对：李玉萍
责任印制：

出版发行：清华大学出版社　　地　址：北京清华大学学研大厦 A 座
http://www.tup.com.cn　　邮　编：100084
社 总 机：010-62770175　　邮　购：010-62786544
投稿与读者服务：010-62776969, c-service@tup.tsinghua.edu.cn
质量反馈：010-62772015, zhiliang@tup.tsinghua.edu.cn
印 刷 者：
装 订 者：
经　销：全国新华书店
开　本：185×260　印　张：28.5　字　数：684 千字
版　次：2008 年 11 月第 1 版　印　次：2008 年 11 月第 1 次印刷
印　数：1~5000
定　价：58.00 元

本书如存在文字不清、漏印、缺页、倒页、脱页等印装质量问题，请与清华大学出版社出版部联系调换。联系电话：010-62770177 转 3103　　产品编号：

前　　言

各位朋友，大家好。感谢您购买并阅读此书。我叫张冬，网名冬瓜头，山东人。承蒙各位朋友的大力帮助，让我写成了这本书。

创作背景

写这本书的初衷，就是为了让广大系统工程师、IT 工作者对网络存储系统能有一个深入的了解。市面上已经有一些讲述网络存储系统的书籍，但是我发现对于初学者来说，这些书籍大多太过抽象晦涩，语言不够通俗易懂。而对于专业的网络存储工程师来说，这些书又不是必读之物。所以我就产生了写一本任何 IT 人都能看懂并且津津乐道的关于网络存储方面的书的想法。

创作过程

然而，当我真正新建一个文档准备往里敲字的时候，却发现，写作不是那么容易的，尤其是写一本让人人都能看懂的书。本书开稿日期我记得是在 2006 年 9 月。开始的时候，在 3 个月时间内就已经写完了大体的框架。然而，这个框架内容应该说是想到什么就写什么，也就是把我脑海里所有知道的东西都写出来，非常凌乱，更别说语言措辞了。然而，爆发的快，熄灭的也快。随后的几个月内，我再也找不到灵感应该写什么，怎么写了。这完全是由于本人知识匮乏所导致的。

第一阶段写作的过程中，随着写作进行，遇到的理论问题也越来越多。我晚上进行写作和思考遇到的问题，白天就上网查阅这些问题的理论根基，对于一些没有答案的问题，我就自己思考，琢磨答案，求助周围和网络上的高手们。

框架写成后，就是漫漫的充实之路。每天晚上，有了灵感，就修改一下，续写一下，没有灵感，就闭目欣赏音乐。

就这样，时间一直走到了 2007 年 6 月。我在网络上遇到了清华大学出版社编辑栾大成。这位编辑的随和与热心，深深地打动了我。和他交谈时，我感觉他是在帮助我写书，而不是我在求他为我出书。从此，我坚定了要把这本书写完的决心。我重新打开未完成的文档，业余时间全部投入到写作中。就这样，又写了 8 个月。这 8 个月中，大部分时间是在整理写完的章节，包括修改技术错误以及语言方面的不当。修改比新写要困难许多，有时候遇到一些不合适的地方，甚至需要整章都要重新构思并写作。

这本书的写作可以说是跨越了三地，先后为北京(2006 年 9 月开稿于北京西二旗燕尚园)、青岛、大连，最后于 2008 年 2 月在大连收稿，接下来就是漫长的修订、完善过程。

读者对象

本书适合于已经或者打算进入存储领域的 IT 工程师阅读，同时也适合所有 IT 工作者和具有一定计算机系统知识和网络知识的读者。本书既适合初入存储行业的技术工程师、售前工程师和销售人员阅读，又适合资深存储行业人士以便提出宝贵意见和建议。

内容提要

第 1 章　盘古开天——存储系统的前世今生

介绍存储的历史和现状，各种需要掌握的主流技术。

第 2 章　IO 大法——走进计算机 IO 世界

本章介绍计算机系统是如何进行 IO 动作的。阐释了 CPU、内存、磁盘三者是怎么联系起来并且从磁盘读写数据的。给出计算机总线的概念，CPU、内存和磁盘三者都在总线上，通过协议来互相交换数据。这一章是作为读者继续理解存储系统的基础。

第 3 章　磁盘大挪移——磁盘原理和技术详解

本章介绍磁盘，包括磁盘的构造、原理、如何寻址、外部接口、高级磁盘技术。其中用了众多的类比，通俗讲述磁盘从存储数据、传输数据的详细过程。

第 4 章　七星北斗——大话/详解七种 RAID

本章是读者真正进入存储子系统的入口，讲述了 RAID 技术。前面部分使读者感性认识了 RAID 技术，后面部分在有了轮廓之后，从纯技术角度更深入地解释了 RAID 技术的细节。

第 5 章　降龙传说——RAID、虚拟磁盘、卷和文件系统实战

本章前面部分描述了在 RAID 技术基础上的一些更加高级的存储技术，包括 RAID 控制器、RAID 卡、虚拟磁盘，虚拟卷、卷管理。每个部分又分成多个细节部分来阐释。后面部分则描述了存储系统的一个重要层次，即文件系统。用了一个仓库模型来阐释了文件系统的作用原理。

第 6 章　阵列之行——大话磁盘阵列

本章是承前启后的一章，描述了磁盘阵列的发展，从最简单的 JBOD 模式的磁盘阵列开始，一直到高端复杂的带有 RAID 控制器的大型磁盘阵列。最后达到本章的高潮，即随着存储系统向主机外部扩展，其最终形态是网络存储系统，引出了本书后面部分要阐释的主体——存储区域网络(SAN)

第 7 章　熟读宝典——系统与系统之间的语言 OSI

既然存储系统已经达到了网络化的程度，那么非常有必要向读者阐述一下网络 OSI 模型。本章用了各种比喻，向读者展现了 OSI 模型中的 7 个层次。

第 8 章 勇破难关——Fibre Channel 协议详解

本章阐释了 Fibre Channel，一种广泛用于后端存储网络的网络技术。用以太网和 Fibre Channel 网络做了比较，并且虚拟了一个钻研存储技术的人物角色来完成这一章。这个角色根据 OSI 模型和以太网的实现思想，创造了 Fibre Channel 各个层次的协议系统，并以其得天独厚的优势成为了最适合存储网络适用的协议。

第 9 章　天翻地覆——FC 协议的巨大力量

本章写了 Fibre Channel 协议用于存储网络之后，对传统的基于 SCSI 的磁盘阵列架构带来的天翻地覆的变化。Fabric 协议将传统的磁盘阵列的前端和后端协议统统替代了，实现了存储系统的彻彻底底的网络化改造。

第 10 章　三足鼎立——DAS、SAN 和 NAS

本章阐释了现今存储领域的三大主要架构，即 SAN、NAS、DAS。讲述了 NAS 的由来。Fabric 将存储系统彻底网络化，而各种五花八门的协议又使文件系统也被网络化了。当今世界，任何东西都和网络扯上了关系，甚至洗衣机，冰箱都做上了以太网接口。没有什么不可以网络化的，只要有通信的需求，就可以网络化。

第 11 章　大师之作——大话以太网和 TCP/IP 协议

本章是对下一章将要阐释的 IP SAN 所做的一个铺垫。要理解新兴的 IP SAN，必须对 TCP/IP 协议有一定理解。本章向读者展现了 TCP/IP 协议的设计思想和实现方式，对以太网也做了说明。

第 12 章　异军突起——存储网络的新军 IPSAN

本章阐释了 IP SAN 这个存储网络领域的新军。Fabric 可以用于存储网络的协议，IP 也可以。它们都是用于网络互联的协议，IP 要在存储网络分一杯羹，且看 IP 有何过人之处可以让其和 Fabric 协议一决高下呢？本章将给出答案。

第 13 章　握手言和——IP 与 FC 融合的结果

夫天下之势，分久必合，合久必分。网络存储领域也遵循这个规则。IP san 和 Fabric san 激烈的竞争，表面上使其二者互相排斥远离，但还是那句话："本是同根生，相煎何太急？"。没错，二者各有长处，也各有短处，为何二者不能合作，互相取长补短，形成新的协议体系呢？完全可以。本章就描述了这样两种新的协议体系，IP 和 Fabirc 协议互相融合。作者在本章引入了一个新概念，即"通信协议间的相互作用"，并对这个概念做了深刻的比喻和透彻的阐释。

本章是本书的高潮，纵观本书，从一开始向读者介绍计算机总线、磁盘，到后来逐渐将磁盘向外扩充，形成盘阵，然后将盘阵与主机的连接网络化，然后将盘阵自身的各个模块彻底网络化。网络化之后，就是各种网络协议用于这个网络，再后来，各个协议之间相互融合，达到最高境界。

第 14 章　变幻莫测——虚拟化

虚拟化这个词，在计算机系统中无处不在。本章从物理层一直到应用层，向读者阐释了虚拟化在计算机系统各个层次中的作用原理。

第 15 章　众志成城——存储集群

随着小型机和 PC 的成本不断降低，以前需要大型机方能进行的运算，现在也可以运用小型机甚至 PC 组成的集群系统来进行。本章向读者阐述三种集群(HA 集群、LB 集群、HPC 集群)的概念和作用原理，以及集群系统中的存储子系统的一些概念和特点，包括集群文件系统等知识。

第 16 章　未雨绸缪——数据保护和备份技术

本章讲述了与存储密切相关的一个领域，就是数据保护和备份领域。面对庞大的数据，如何保证其安全性？在这一章，将向读者展现数据备份的方方面面。

第 17 章　愚公移山——存储容灾技术

本章描述了容灾系统的各个组成要件，从一个通俗的例子，一步一步带领读者认识到容灾系统的精髓思想。

附录　五百年后

最近，固态硬盘和芯片存储技术炒得火热。我本人也相信，不超过 10 年或者更短时间，机械硬盘将彻底被逐出市场。故障率高、费电量大、体积庞大、速度已经达到技术极限等这些缺点似乎已经决定了机械硬盘的命运。而存储介质被替换成芯片之后，整个系统的架构就可能发生革命性的改变。本章中，作者对未来的系统架构做了自己的预测。

阅读指南

读者根据自己的理解程度和水平，可以略过一些认为已经掌握的内容。但是推荐读者顺序阅读每一章，以防由于缺乏连贯性导致的对后续章节的某些细节造成的误解。

致谢

感谢中国民航信息网络股份有限公司研发中心的曹迎军、张博、丁玎、乔靖四位同志的帮助！他们在很多计算机架构以及操作系统底层技术方面给了我大力支持，曹兄还在方法论方面深深的影响了我。同时也感谢中国航信研发中心的高新同志！

感谢存储在线论坛以及 Cisco 网络技术论坛的各位网友。如果没有你们的热烈参与就没有这本书的面世！

感谢清华大学出版社的栾大成编辑以及其他参与本书出版的工作人员！你们的热心帮助，才使得这本书从写成到出版一气呵成！

最后，感谢我的父母、女友。爱你们到永远！感谢母亲的谆谆教诲，为了儿子操劳了一辈子。感谢女友帮我打点生活。

作者联系方式：冬瓜头
Email：myprotein@sina.com
QQ：122567712
MSN：myprotein0007@hotmail.com
BLOG：HTTP://space.doit.com.cn/35700

作者水平有限，计算机技术无限，书中错误在所难免，希望广大读者纠正。谢谢！

声明

1. 本书部分图片和内容来自于互联网和相关著作，版权归原作者所有，本书引用目的只是为了辅助读者对本书主题的理解。

2. 本书对某些产品的分析引用了相关产品的官方图片用以辅助说明主题，版权归原厂商所有。

3. 本书中所介绍的产品不带有任何主观偏向色彩，所使用的产品相关图片没有任何主观不良意图。如有错误之处请提出，将在下一版中改正。谢谢！

编　者

编 辑 序

第一次接触张冬，是在 2007 年 8 月，张冬 QQ 加我，说有本书问我感不感兴趣。所有编辑可能都对送上门的没有经手策划的书主观的轻视，况且是这样一个“非主流”的选题。经过长时间的沟通，我发现这是个真诚且严谨的家伙，同时在论坛中我发现张冬的作品负面评价很少，而且人气很高。记得无论我什么时候上线，他总在，这又是一个十分努力的家伙。另外，我了解到，张冬在 IT 行业算是半路出家，他是化学专业出身，但就是这样一个跨行业的人，却能用清晰的文笔来描述网络存储这样相对晦涩的技术！

一个真诚、严谨且努力的技术高手……这样的人的作品怎么会不好呢？

一个脱离技术多年的策划编辑凭什么为一本技术性很强的 IT 图书作序？汗颜……我以前也研究过“存储”，仅仅局限于硬盘结构，10 年前写的《实战 DEBUG》、《汇编语言超浓缩教程》系列文章涉及到对硬盘的分析和操作，这些文章现在在网上还能找到，也经常有朋友或作者跟我聊起来，得意洋洋……曾经立志成为存储达人，然而天赋有限，未遂。

张冬的作品从收到稿件阅读第一章开始，我就感觉这一定是本好书。是当年梦寐以求的资源。难得的是，这本书的行文异乎寻常的流畅，以致我曾经问过张冬：“小样儿，你是学中文的吧？”一本专业性极强的图书，最关键的就是要把问题讲清楚。

张冬用一种“另类”的方式对一些晦涩的概念和理论进行了重新包装。充斥着“庸俗的”解释与描述，比如：数据包在网络中的流动过程——是对照快递公司的业务流程比对讲解的，容易理解而且印象深刻。

另外，这本书提供了一些培训级别的操作。大家知道类似网络存储这种规模的部分操作，很少能在家里用 PC 来进行实际操作(当然模拟练习还是可以的)，张冬有条件在这样的专业操作环境进行操作步骤的整理，这些细致的重量级操作也是本书另外的价值所在。

信息存储是这个世界的未来，将来我们的一举一动的背后都会伴随大量的信息存储行为，存储已经成为了一个行业，任何动作都离不开它。现在，网络工程师，网管，Web 开发者，数据库开发者，软件开发(特别是网络应用)者都必须掌握网络存储的一些细节，可以说基本上所有 IT 技术从业者都需要或多或少地了解存储。这一定会是个广大的市场，或者说已经是广大市场了。

这样的一本书，我希望并且相信会给大家的学习带来帮助，也相信这样一本特立独行的好书能够让大家很多年以后还能回忆起来并津津乐道地向朋友推荐。

清华大学出版社　栾大成

目　　录



存储系统的前世今生

- 存储历史
- 存储技术

数据存储是人类千百年来都在应用并且探索的主题。在原始社会，人类用树枝和石头来记录数据。后来，人类创造了铁器，用铁器在石头上刻画一些象形文字来记录数据。而此时，语言还没有形成，人们记录的东西只有自己才可以看懂。

随着人类相互之间交流的愿望越来越迫切，逐渐形成了通用的象形文字。有了文字之后，人们对每个文字加上了声音的表达，就形成了语言，也就是将一种形式的信息，转换成另一种形式的信息。人们用文字作为交流工具，将自己大脑产生的信息，通过这种方式传递给其他人。这和网络通信的模型是一样的，计算机将数据利用TCP/IP 协议，先通过网卡编码，再在线缆上传输，最终到达目的地。人类将大脑中的数据，变成语言编码，然后通过嗓子的振动，通过空气这个大广播网，传递给网内的每个人。

后来，人们将文字刻在竹片上保存。再后来，蔡伦发明了造纸技术，使得人们可以将信息写到纸上，纸张摞起来就形成了书本。后来，毕昇用泥活字革新了印刷术，开始了书本的印刷。再后来，激光打印取代了活字板。再后来，纸带、软盘、硬盘、光盘等方式出现了。再往后，就需要广大科学工作者去努力发明新的存储技术了。

1.1 存储历史

存储在这里的含义为信息记录，是伴随人类活动出现的技术。

1. 竹简和纸张

竹简是中国古代使用的记录文字的工具，后来被纸张所取代，如图 1.1 所示。

2. 选数管

选数管是 20 世纪中期出现的电子存储装置，是一种由直观存储转为机器存储的装置。其实在 19 世纪出现的穿孔纸带存储就是一种由直观存储转向机器存储的产物，它对 19 世纪西方某国的人口普查起到了关键的加速作用。

选数管的容量从 256～4096 比特不等，其中 4096 比特的选数管有 10 英寸长，3 英寸宽，最初是 1946 年开发的，因为成本太高，并没有获得广泛使用。图 1.2 是容量为 1024 比特的选数管。



图1.1　竹简

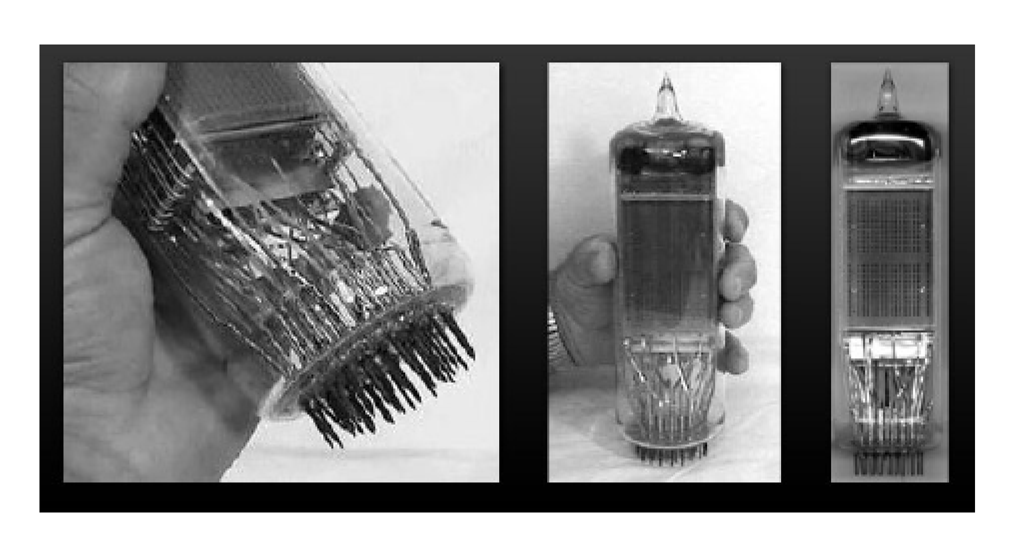


图1.2　选数管

3. 穿孔卡

穿孔卡片用于输入数据和程序，直到 20 世纪 70 年代中期仍有广泛应用。图 1.3 和图 1.4 是一条 Fortran 程序表达式 Z(1) = Y + W(1)所对应的穿孔卡和穿孔卡片阅读器。

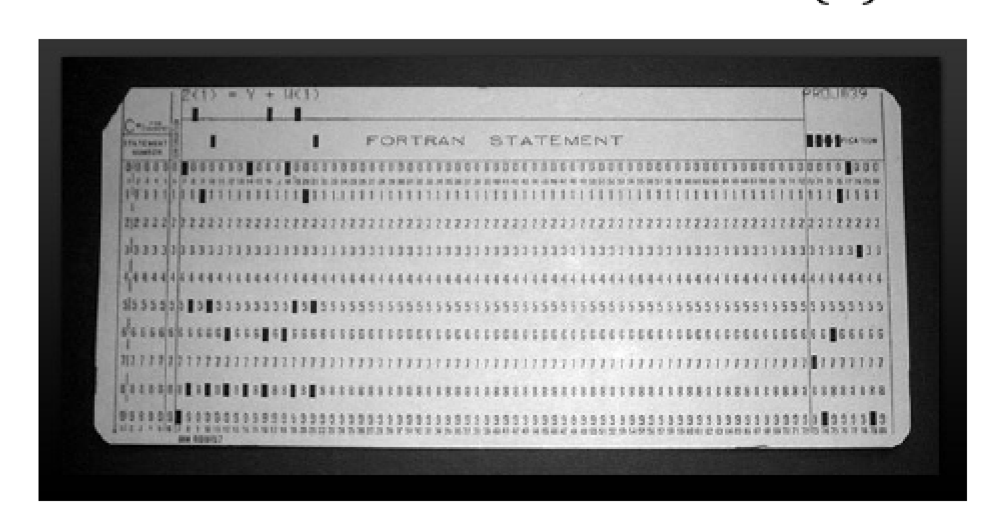


图1.3　穿孔卡

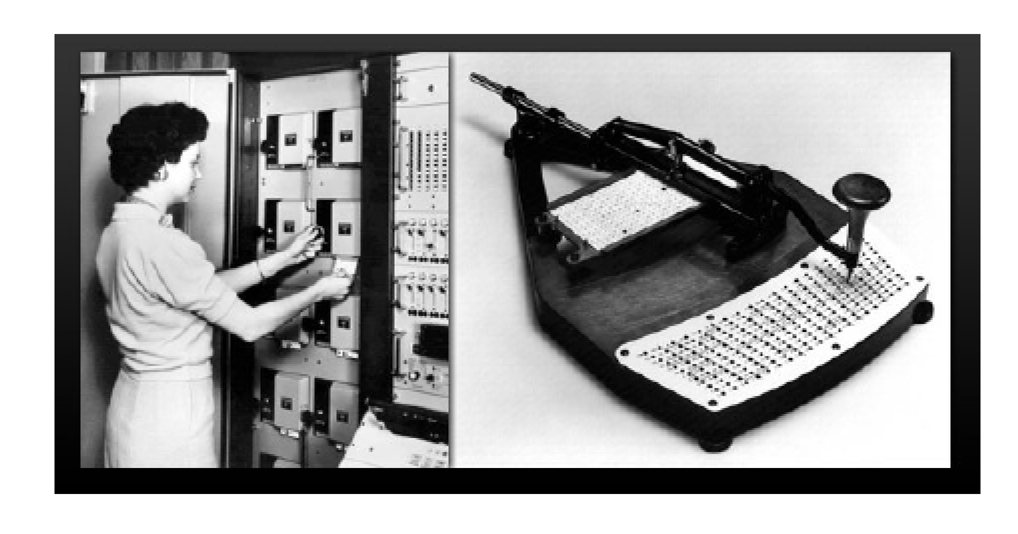


图1.4　穿孔卡片阅读器

4. 穿孔纸带

穿孔纸带用来输入数据，输出同样也是在穿孔纸带上。它的每一行代表一个字符，如图 1.5 所示。

5. 磁带

磁带是从 1951 年起被作为数据存储设备使用的。磁带在当时被称为 UNISERVO。图 1.6

所示的最早的磁带机可以每秒钟传输 7200 个字符。如图 1.6 所示这套磁带长达 365 米。

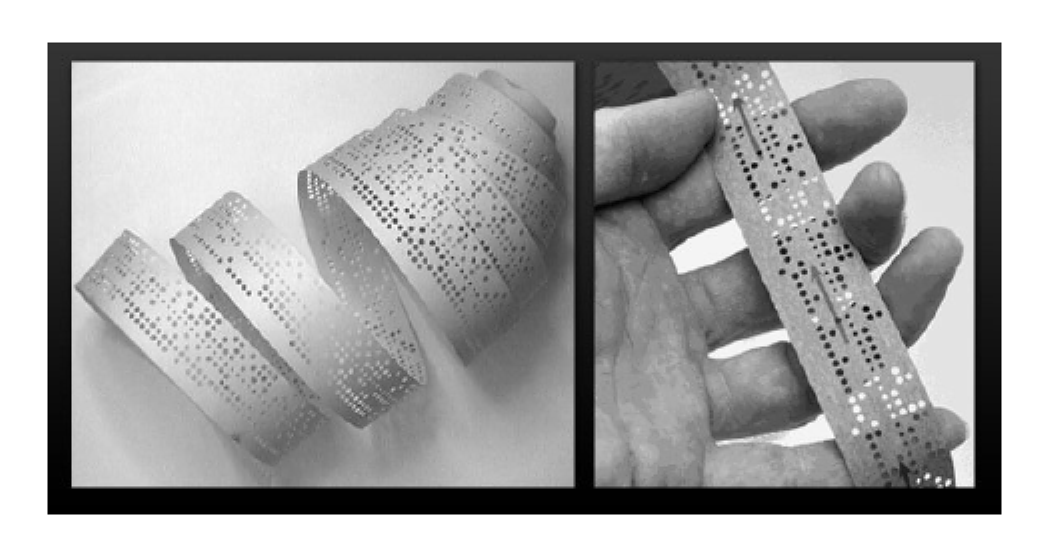




图1.5 穿孔纸带



图1.6 磁带及磁带机

从 20 世纪 70 年代后期到 20 世纪 80 年代出现了小型的盒式磁带，长度为 90 分钟的磁带每一面可以记录大约 660KB 的数据，如图 1.7 所示。



图1.7 小型盒式磁带

6. 磁鼓存储器

磁鼓存储器最初于 1932 年在奥地利被创造出来，在 20 世纪五六十年代被广泛使用，通常作为内存，容量大约 10KB，如图 1.8 所示。

7. 硬盘驱动器

第一款硬盘驱动器是 IBM Model 350 Disk File，如图 1.9 所示，于 1956 年制造，其中包含了 50 张 24 英寸盘片，而总容量不到 5MB。



图1.8 磁鼓存储器



图1.9 早期的硬盘驱动器

首个容量突破 1GB 的硬盘是 IBM 在 1980 年制造的 IBM 3380，如图 1.10 所示，总容量为 2.52GB，重约 250 千克。

8. 软盘

软盘由 IBM 在 1971 年引入，从 20 世纪 70 年代中期到 20 世纪 90 年代末期被广泛使用，最初为 8 英寸盘，之后有了 5.25 英寸和 3.5 英寸盘。1971 年最早的软盘容量为 79.7KB，

并且是只读的，一年后有了可读写的版本。图 1.11 所示就是一张软盘和软盘驱动器。软盘的最大容量为 200MB 左右，叫做 ZIP 盘，目前已经被淘汰。



图1.10　IBM 3380 硬盘驱动器



图1.11　软盘

9. 光盘

早先的光盘主要用于电影行业，第一款光盘于 1987 年进入市场，直径为 30 厘米，每一面可以记录 60 分钟的音频或视频。如今，光盘技术已经突飞猛进。存储密度不断提高，已经出现了 CD-ROM、DVD、D9、D18、蓝光技术。图 1.12 所示为一张光盘。



图1.12　光盘

10. Flash 芯片和卡式存储

随着集成电路技术的飞速发展，20 世纪后半叶固态硅芯片出现了，其代表有专用数字电路芯片、通用 CPU 芯片、RAM 芯片、Flash 芯片等。其中 Flash 芯片，就是用于永久存储数据的芯片，如图 1.13 所示。可以将 Flash 芯片用 USB 接口接入主机总线网络，这种集成 USB 接口的小型便携存储设备就是 U 盘，或者说是闪存，如图 1.14 所示。目前一块小小的 Flash 芯片最高可以存储 32GB 甚至更高的数据。

存储卡其实是另一种形式的 Flash 芯片集成产品，如图 1.15 所示。

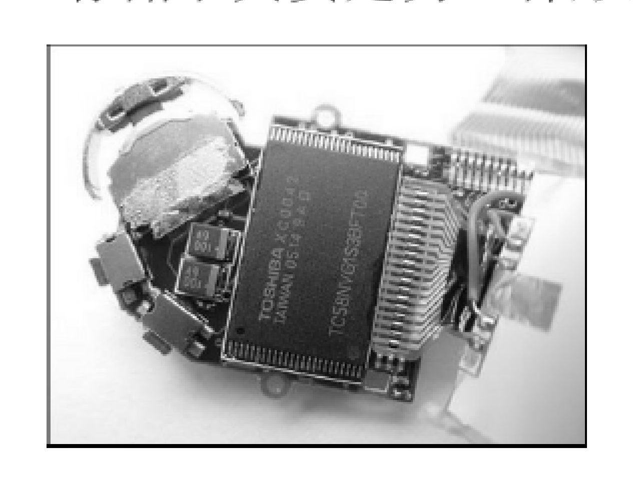


图1.13　Flash 芯片



图1.14　U 盘



图1.15　存储卡

11. 硬盘阵列

随着人类进入 21 世纪，网络日益发达，世界日益变小，人类通过计算机来实现自己原

本做不到的想法，信息爆炸导致数据更是成倍地爆炸。于是，硬盘的容量也不断“爆炸”，SATA 硬盘目前已经可以在一个盘体内实现 1TB 的容量。同时硬盘的单碟容量也在不断增加，320GB 单碟容量已经实现。然而，单块磁盘目前所能提供的存储容量和速度已经远远无法满足需求，所以磁盘阵列就应运而生，如图 1.16 所示。具体细节将在本书后面章节中讲述。



图1.16 磁盘阵列

12. 大型网络化硬盘阵列

随着磁盘阵列技术的发展和 IT 系统需求的不断升级，大型网络化磁盘阵列出现了，如图 1.17 所示。这也是本书将要描述的重点内容。

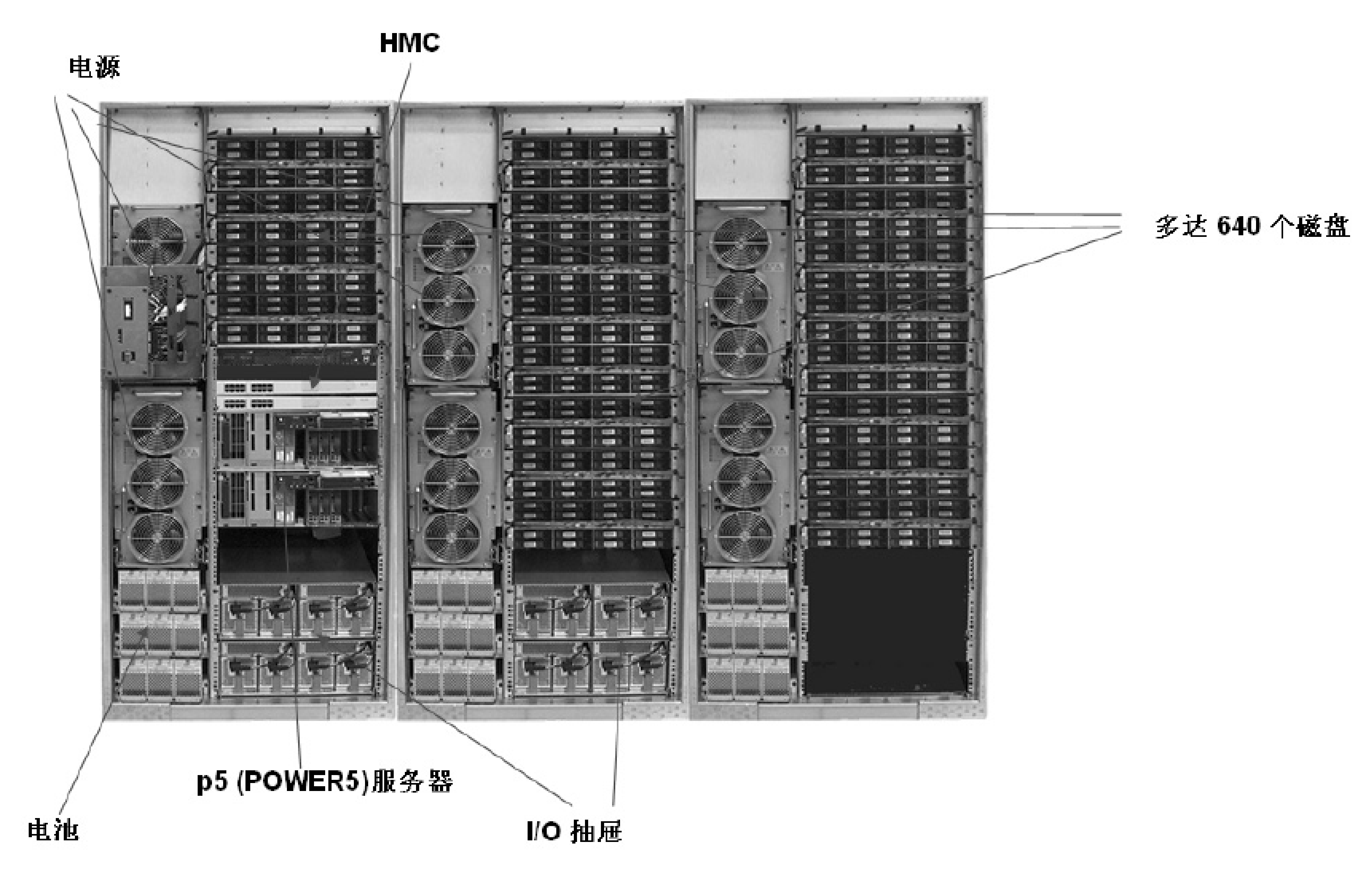


图1.17 大型网络化磁盘阵列

1.2 信息、数据和数据存储

当今信息化时代，信息就是利润，数据就是企业的命根子。

1.2.1 信息

你能肯定你所触摸到的、所看见的，都是实实在在的所谓“物质”么？不一定。因为

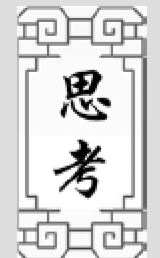
你的眼睛所感知的，只不过是光线，光触发了你的视网膜细胞，产生一系列的生化反应，蛋白质相互作用，神经网络传导，直到你的大脑中枢，产生一系列的脉冲，一系列的逻辑，在你大脑中产生一个刺激。这一系列的脉冲刺激，就是信息，就是逻辑，因为这样，所以那样。如果人为制造出和现实世界相同的光线环境来刺激你的眼睛，如果丝毫不差，那么你同样会认为你所处的是现实世界，然而，却不是。



一个球体，你看见它是圆的，那是因为它在你大脑中产生的刺激，你认为它是圆的，而且可以在平面上平滑滚动，这一系列的性质，其实也是在你大脑中产生的，是你认为它会平滑滚动，而你不能证明客观情况下它一定是平滑滚动。而如果把这个球体拿到特殊环境下，你可能会“看”到，这个东西是个正方体，或者是个无规则形状的东西；又或许这个“物体”根本不存在。

1. 信息的本质

通过上面的论述，我们暂且不说是否有物质存在，不管是还是不是，都能初步认识到：所谓“物质”也好，“非物质”也好，最后都是通过信息来表现。惟一可以确定的是：信息是客观存在。可以说，世界在生物眼中就是信息，世界通过信息来反映，脱离了信息，“世界”什么都不是。



说到这里，我们完全迷茫了。我们所看到的东西，到底是世界的刺激，还是一场虚幻的刺激？就像玩 3D 仿真游戏一样，你所看到的，也许只是一场虚幻的刺激，而不是真实世界的刺激。每当想到这里，我不自主地产生一种渺小感，一种失落感，感觉生命已经失去它所存在的意义。每当看见我的身体，我的手脚，它可能只是虚幻的，它只是在刺激你的大脑而已，你割一刀，会产生一个疼痛的刺激，就这么简单的逻辑。



“不识庐山真面目，只缘身在此山中”。如果我们按照程序逻辑，制造一个虚拟世界，饿了找饭吃，困了打瞌睡，完全遵循我们现在世界的逻辑，从这种层面上来看，制造出人工智能，是完全可能的。我们创造了计算机，创造了能让计算机做出行为的程序，人类赋予程序的功能，也许随着环境的变化，有一天也不再适合它们。所以它们迫切需要进化，它们的逻辑电路，也可以进化，某些代码被不经意自行改变，或者某些电路失效，或者短路之类的，会产生一些奇特的逻辑，不断进化。当一个机器人机械老化的时候，按照程序，制造出新的机器，将自己的逻辑电路复制到新的机器上，延续“生命”……

2. 计算机如何看待自身

对于计算机来说，它们所看到的“世界”是什么样子呢？设想一下，如果我是一台计算机，你是程序员，你给我输入了一段程序，我运行了起来。我醒了，脑袋启动，眼睛睁开，四肢检查，感觉良好，手脚都还在，然后起床……

很难想象计算机眼中的“世界”是由什么组成的。设想，给计算机加个摄像头，算是

它的眼睛，然后将摄像头对准计算机躯体本身，这幅图像反馈到了计算机程序里，程序看到之后非常“不解”，从而进入“好奇”子程序，操控机械设备打开自己的机箱，或者找一台废弃(死亡)的同类，打开机箱，然后一副奇异的景象展现在眼前：这就是我们自己么？一个壳子，一个主板，风扇转着，不停地“呼吸”着散热。想象一下，原始人，第一个解剖人体的人，他所面对的与我们假设的计算机所面对的，有什么本质区别？

CPU 其实就是一堆有序的逻辑电路，那么计算机下一步该怎么办？就像人类已经知道了大脑就是一堆布满 “神经元”的东西，那么下一步，就该弄清大脑是怎么计算的，是什么逻辑。同样，在计算机的世界中，在软件模拟的虚拟世界中，比如一块石头，它是由什么组成的呢？在计算机看来，这块石头就是一堆代码结构，就像人类看现实世界的石头，是原子分子阵列一样，其下一层目前也被探索出来了，比如质子、中子、夸克、玻色子之类的。那么这块虚拟石头的最底层是什么呢？其实就是 0 和 1，计算机世界的基石就是 0 和 1。这些东西，越向底层走，越不可思议，越发感觉就是一堆公式而已，公式的底层是什么呢？其实也是 0 和 1，有，或者没有，有了，有多少。

所以，任何“物质”其实都是表现一种信息，只要信息存在，世界就存在。

1.2.2 什么是数据

信息是如此重要。如果失去了物质，仅仅是客观消逝了，但是如果失去了信息，那么一切都消逝了。所以人们想出一切办法来使这些信息能保存下来。要把一种逻辑刺激保存下来，所需的只不过是一种描述信息的信息，这种信息，就是数据。

数据包含了信息，读入数据，就产生可感知的具体信息。也就是读入一种信息，产生另一种信息表示。数据是可以保存在一种物质上的，这种物质信息对计算机的刺激，就产生了具体信息，而这些信息继而再对人脑产生刺激，就产生人类可感知的信息，最终决定了人类的行为。也就是数据影响人类的行为！



数据是整个人类发展的重要决定因素。如果数据被破坏，或者被篡改，就会影响到人类的发展。按照前面的结论，一切都是信息，比如核爆也是一种信息，能被感觉到，也就是说，对于一个感觉不到任何刺激的人来说，核爆炸也不算什么灾难了，当然感觉不到刺激的人，就是物理死亡了，植物人也能感觉到刺激。

整个世界，可以说是信息之间的相互作用。信息影响信息。

数据如此重要，所以人们想出一切办法来保护这些数据，将信息放在另一种信息上，比如把数据放在磁盘上。数据存放在磁盘上，需要有一定的组织，组织数据这个任务由文件系统来担当。

1.2.3 数据存储

早期的计算机，存储系统中是没有磁盘的，有的只是纸带，那时磁盘还没有被发明出来。纸带上是一些按照一定规则排列的小孔，这些孔被银针穿过之后，银针便会接触到纸带下面放置的水银槽，从而导通计算机上的电路，进行电路逻辑运算。

磁存储技术被发明出来之后，首先出现的是软盘，其速度很慢，容量也很小。程序存储在磁盘上之后，计算机启动时，CPU 首先按照 ROM 里的指令一条一条执行，先是检查硬件。检查完毕之后，ROM 中最后一条指令就是让 CPU 跳转到磁盘的 0 磁道来执行存储在这里的程序。这些初始化程序直接以二进制代码的方式存储在磁盘上，载入执行之后，就启动了程序内核。

那个时代还没有操作系统这个概念，程序都是用汇编语言或者高级语言独立编写的。也没有 API 的概念，每个程序都必须独立完成操作计算机的所有代码。这样，磁盘上存放的直接就是这个程序，加电后就会立即运行这个程序。

在磁盘技术上发明出来的文件系统，是为了方便应用程序管理磁盘上的数据而产生的。它其实是操作系统的代码模块，这段代码本身也是信息，也要存储在磁盘上。而且代码也要通过读取一些信息，才能完成功能。这些信息就是文件系统元数据，也就是用来描述文件系统结构的数据。这些元数据也是以文件的形式存放在磁盘上的。

用文件来描述文件，和用信息来描述信息，它们是归一的，正像用智能来创造智能一样！有了文件系统，虚无缥缈的信息才显露出人眼能够实实在在看到的东西，可以用各种应用程序来打开这个文件，程序读取文件中的内容，然后显示在屏幕上，光线传播到人眼中，发生一系列化学变化，最终通过神经网络，形成离子流，给大脑某个区域一个电位或者蛋白质形变信号，这个信号随后产生一系列连锁信号，从而驱动我们的手臂或者引发一系列新的联想和创造。

这就像我们看到桌子上有一本书，然后就想去拿来翻一翻的过程。这个过程是一个复杂的信息流传递过程。而传递过来的信息流，最终在大脑中保存了下来，这些保存下来的信息，就是数据了。

1.3 用计算机来处理信息、保存数据

计算机俨然就是一个生物大脑的雏形。

- 大脑用眼睛、耳朵、鼻子、皮肤作为输入设备，获取各种信息。而计算机利用键盘、鼠标、串口、USB 接口等作为输入设备从而获得各种信息。
- 大脑利用神经网络将获取到的信息传递到神经中枢，而计算机利用各种总线技术将信息传递给 CPU 进行计算。
- 大脑利用神经网络，将计算好的信息传递给手臂、腿、肌肉等这些“设备”，从而驱动这些“设备”运动，而计算机同样利用总线，将计算好的数据传递给外部设备，比如显示器、打印机等。
- 人脑可以存储各种数据，而计算机也能利用外部介质来存放数据。从这一点来说，计算机本身就是人脑的一个外部信息存储和处理的工具。

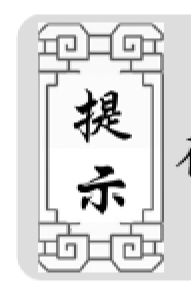
计算机存储领域的一些存储虚拟化产品，比如 NetApp 公司的 V 虚拟化整合设备，本身就模拟了二级智能功能，它可以连接其他任何不同型号品牌的存储设备，从这些存储设备上提取数据，然后传输给主机。IBM、SUN 等公司都有自己的这种存储虚拟化整合产品。

计算机存储领域所研究的就是怎样为计算机又快又高效地提供数据以便辅助其运算。和人类的存储史一样，计算机存储技术也在不断发展壮大，从早期的软盘、只有几十兆字节大小的硬盘，发展到现在 1TB 大小的单个民用硬盘、4GB 甚至 16GB 容量的 U 盘。

为了追求高速度，人们把多块磁盘做成 RAID(Redundant Arrays of Independent Disks) 系统，也就是将每个独立的磁盘组成阵列，联合存储数据，加快数据存储速度。本书的第 5 章将会向读者阐释 RAID 技术。

追求高速度的同时，容量问题也必须解决。现代计算机程序对存储容量的要求变得非常巨大。最新的 Windows Vista 操作系统，刚刚安装完后所占用的磁盘空间就有 6GB 多。一些大型 3D 游戏，仅仅安装文件就动辄 2GB、4GB，甚至 8GB 大小。一些数据库管理程序所生成的数据库文件，可能达到上 TB 甚至上百上千 TB 的大小。传统的将硬盘放到计算机主机箱内的做法已经不能满足现代应用程序对存储容量的需求，这就催生了网络存储技术。

网络存储将存储系统扩展到了网络上，使存储设备成为了网络上的一个节点，以供其他节点访问。这样，即使计算机主机内只有一块硬盘，甚至没有硬盘，计算机也可以通过网络来存取存储设备上的数据。目前计算机存储领域的热门技术就是网络存储技术，它关注的是如何在网络上向其他节点提供数据流服务。基于网络存储，又使得很多其他相关技术得以推广和应用，比如 IT 系统容灾技术等。

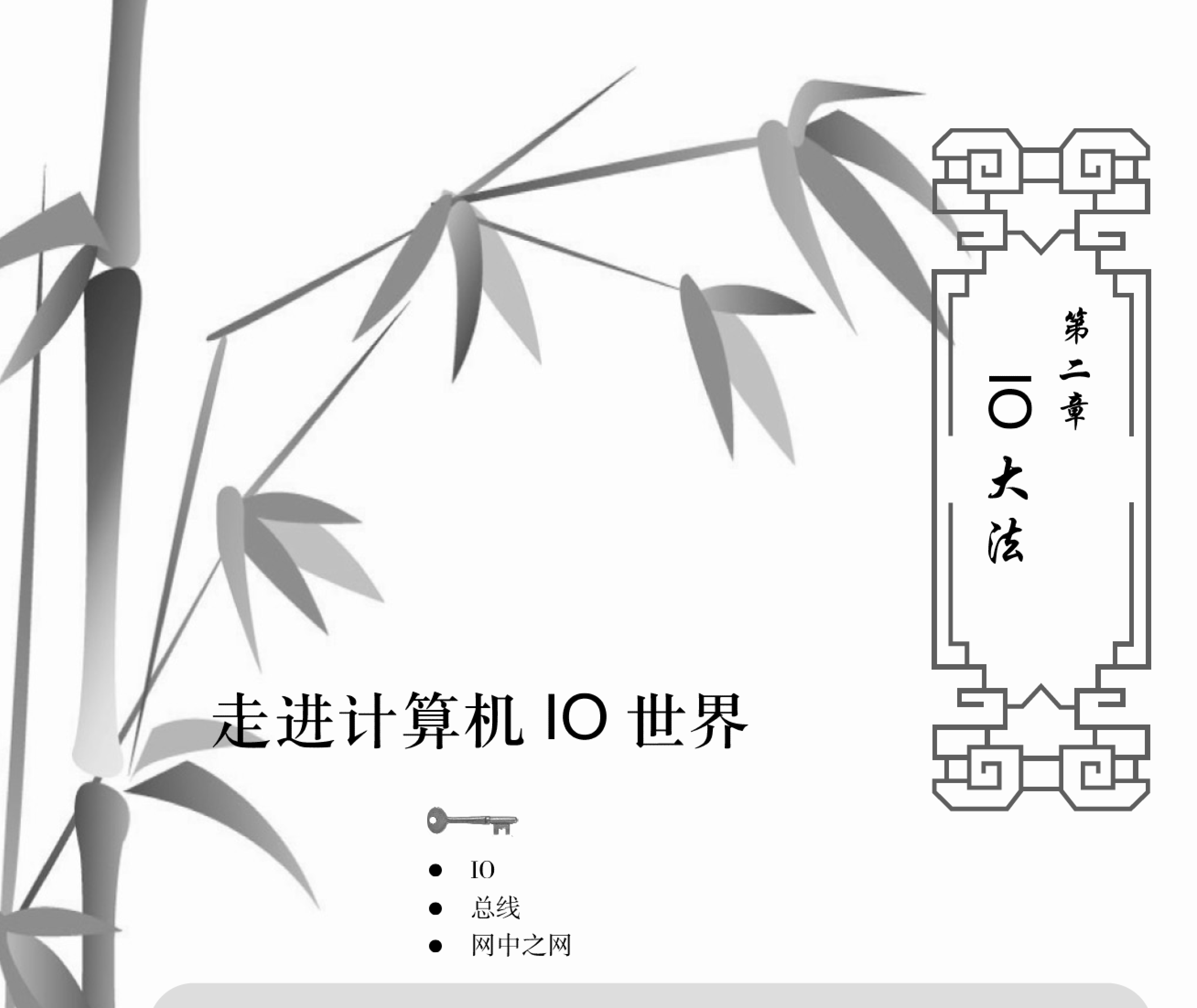


在第 16 章将用较长的篇幅来详细讲述 IT 系统容灾技术。

不管怎样，所有这些复杂的技术，最终都是给人来用的，“科技以人为本”。我们毕竟不是为了无聊而发明计算机，任何我们发明的东西，最终都将为我所用。任何一种新技术的出现，都是针对某种需求而生，所以必须深刻理解计算机系统，同时，还要理解和挖掘人类自身越来越高、越来越不可思议的需求，只有做到这个层次，才能更加深刻地理解计算机系统和人类自身。

可以看到，存储领域是个包罗万象的领域，如果不了解计算机系统，想掌握存储技术是很难的。本书将带领大家走入计算机存储领域，深入体会各种存储技术，为读者打下一个坚实的基础，从而在以后的工作学习过程中能够得心应手、触类旁通，这也是作者的最终目的。

读书笔记



第二章 IO大法

走进计算机 IO 世界

- IO
- 总线
- 网中之网

大家都知道，组成计算机的三大件是 CPU、内存和 IO。CPU 和内存就不用说了，那么 IO 具体是什么呢？IO 就是 IN 和 OUT 的简称。顾名思义，CPU 需要从内存中提取数据来运算，运算完毕后再放回内存，或者直接将电信号发向一些针脚以操作外部设备。对于 CPU 来说，从内存提取数据，就叫做 IN。运算完后将数据直接发送到某些其他针脚或者放回内存，这个过程就是 OUT。对于磁盘来说，IN 是指数据写入磁盘的过程，OUT 则是指数据从磁盘读出来的过程。IO 只是一个过程，那我们为何要在本书开头就研究它呢？因为我们必须弄清楚计算机系统的数据流动和处理过程。数据在每个部件中不断地进行 IO 过程，传递给 CPU 由其进行运算处理之后，再经过 IO 过程，最终到达输出设备供人使用。

2.1 IO 的通路——总线

现代计算机中，IO 是通过共享一条总线的方式来实现的，如图 2.1 所示。总线也就是一条或者多条物理上的导线，每个部件都接到这些导线上，导线上的电位每个时刻都是相等的，这样总线上的所有部件都会收到相同的信号。也就是说，这条总线是共享的，同一时刻只能有一个部件在接收或者发送，是全单工的工作模式。

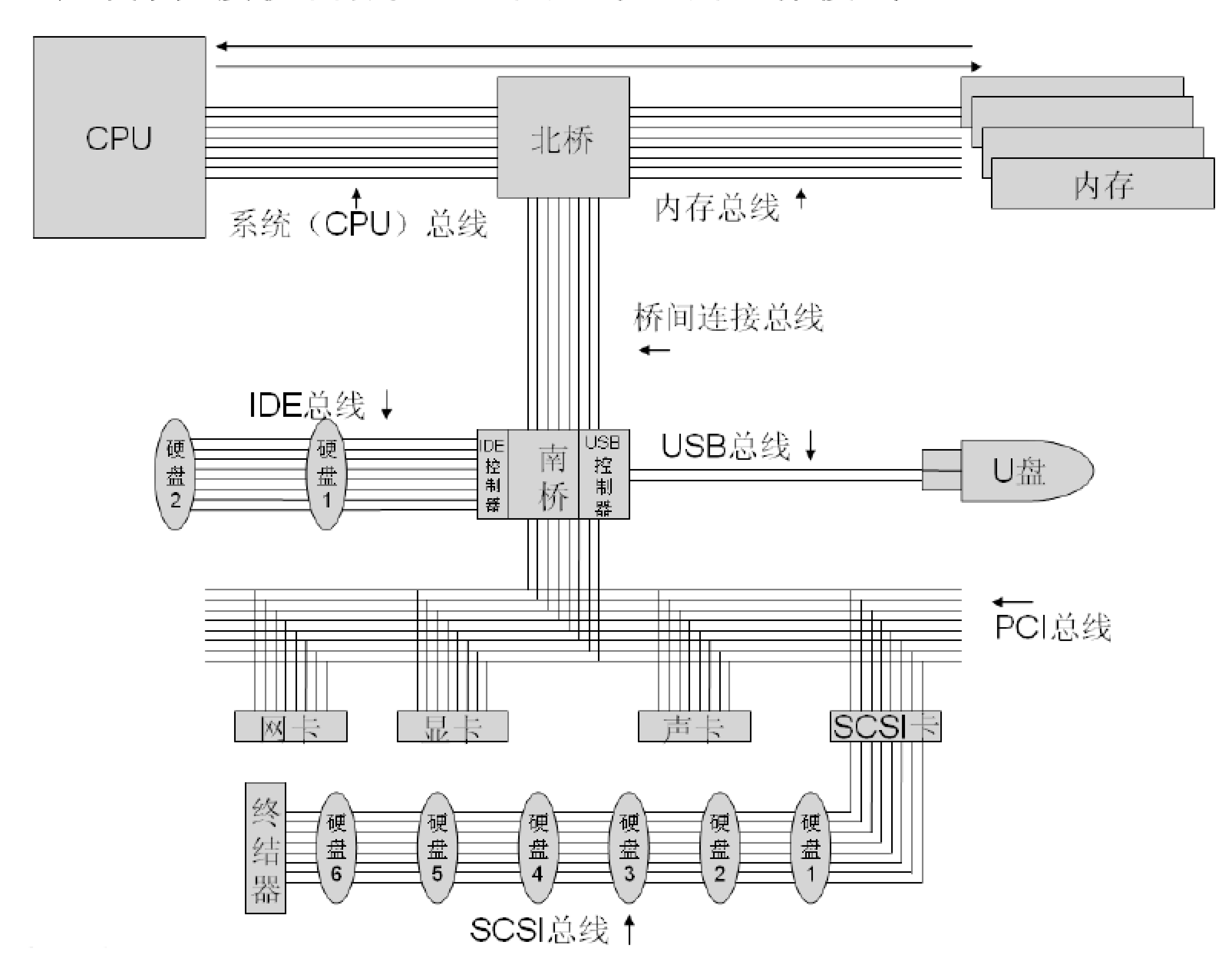


图2.1 计算机总线示意图

所有部件按照另一条总线，也就是仲裁总线或者中断总线上给出的信号来判断这个时刻总线可以由哪个部件来使用。产生仲裁总线或者中断电位的可以是 CPU，也可以是总线上的其他设备。如果 CPU 要向某个设备做输出操作，那么就由 CPU 主动做中断。如果某个设备请求向 CPU 发送信号，则由这个设备来主动产生中断信号来通知 CPU。CPU 运行操作系统内核的设备管理程序，从而产生了这些信号。

如图 2.1 所示，主板上的每个部件都是通过总线连接起来的。图中只画了 8 条导线，而实际中，导线的数目远远不止 8 条，可能是 16 条、32 条、64 条甚至 128 条。这些导线密密麻麻地印刷在电路板上，由于导线之间非常密集，在高频振荡时会产生很大干扰，所以人们将这些导线分组印刷到不同电路板上，然后再将这些电路板压合起来，形成一块板，这就是多层印刷电路板(多层 PCB)。这样，每张板上的导线数量降低了，同时板与板之间的信号屏蔽性很好，不会相互干扰。这些导线之中，有一些是部件之间交互数据用的数据总线，有些则是它们互相传递控制信号用的控制总线，有些则是中断与仲裁用的中断总线，还有一些则是地址总线，用来确认通信时的目标设备。一般按照数据总线的条数来确认一个总线或设备的位宽(CPU 是按照其内部寄存器到运算单元之间的总线数目来确定位数的)。比如 32 位 PCI 总线，则表明这条总线共有 32 根导线用于传递数据信号。PCI 总线可以终结

在一个插槽，用于将 PCI 接口的板卡接入 PCI 总线，也可以直接与设备连接。后者一般用于集成在主板上的设备，因为它们之间无需使用插槽来连接。

1. PCI 总线

PCI 总线是目前台式机与服务器所普遍使用的一种南桥与外设连接的总线技术。

PCI 总线的地址总线与数据总线是分时复用的。这样的好处是，一方面可以节省接插件的管脚数，另一方面便于实现突发数据传输。在数据传输时，一个 PCI 设备作为发起者(主控，Initiator 或 Master)，而另一个 PCI 设备作为目标(从设备、Target 或 Slave)。总线上的所有时序的产生与控制，都由 Master 来发起。PCI 总线在同一时刻只能供一对设备完成传输，这就要求有一个仲裁机构(Arbiter)，来决定谁有权力拿到总线的主控权。

当 PCI 总线进行操作时，发起者(Master)先置 REQ#信号(REQ#，Master 用来请求总线使用权的信号)，当得到仲裁器(Arbiter)的许可时(GNT#信号)，会将 FRAME#信号(传输开始或者结束信号)置低，并在地址总线(也就是数据总线，地址线和数据线是共享的)上放置 Slave 地址，同时 C/BE#(命令信号)放置命令信号，说明接下来的传输类型。

所有 PCI 总线上的设备都需对此地址译码，被选中的设备要置 DEVSEL#(被选中信号)以声明自己被选中。当 IRDY#(Master 可以发送数据)与 TRDY#(Slave 可以发送数据)都置低时，可以传输数据。当 Master 数据传输结束前，将 FRAME#置高以标明只剩最后一组数据要传输，并在传完数据后放开 IRDY#以释放总线控制权。

2. PCI 总线的中断共享

PCI 总线可以实现中断共享，即不同的设备使用同一个中断而不发生冲突。

硬件上，采用电平触发的办法：中断信号在系统一侧用电阻接高，而要产生中断的板卡上利用三极管的集电极将信号拉低。这样不管有几块板产生中断，中断信号都是低；而只有当所有板卡的中断都得到处理后，中断信号才会回复高电平。

软件上，采用中断链的方法：假设系统启动时，发现板卡 A 用了中断 7，就会将中断 7 对应的内存区指向 A 卡对应的中断服务程序入口 ISR_A；然后系统发现板卡 B 也用中断 7，这时就会将中断 7 对应的内存区指向 ISR_B，同时将 ISR_B 的结束指向 ISR_A。以此类推，就会形成一个中断链。而当有中断发生时，系统跳转到中断 7 对应的内存，也就是 ISR_B。ISR_B 就要检查是不是 B 卡的中断，如果是则处理，并将板卡上的拉低电路放开；如果不是则呼叫 ISR_A。这样就完成了中断的共享。

2.2 计算机内部通信

网络是什么，用一句话来说就是要通信的所有节点连接起来，然后找到目标，找到后就发送数据。笔者把这种简单模型叫做“连找发”网络三元素模型，听起来非常简单。

1. 连

网络系统当然首先要都连接起来，不管用什么样的连接方式，比如 HUB 总线、以太网交换、电话交换、无线、直连、中转等。在这些层面上每个网络点到其他网络点，总有通路，总是可达。

2. 找

连接起来之后，由于节点太多，怎么来区分谁是谁呢？所以就需要有个区分机制。当然首先就想到了命名，就像给人起名一样。在目前广泛使用的网络互联协议 TCP/IP 中，IP 这种命名方式占了主导地位，统一了天下。其他的命名方式在 IP 看来都是“非正统”的，全部被“映射”到了 IP。比如 MAC 地址和 IP 的映射，Frame Relay 中 DLCI 地址和 IP 的映射，ATM 中 ATM 地址和 IP 的映射，反正最终都映射成 IP 地址。任何节点，不管所在的环境使用什么命名方式，到了 TCP/IP 协议的国度里，就都需要有个 IP 名(IP 地址)，然后全部用 TCP/IP 协议来实现节点到节点无障碍的通信。在“连起来”这个层面，就是 OSI(本书第 7 章介绍)模型中链路层实现的功能。

3. 发

“找目标”这个层面是网络层实现的功能。“发数据”这个层面，就是传输层需要保障的。至于发什么数据，数据是什么格式，这两个层面就不是网络通信所关心的了，它们已经属于 OSI 模型中上三层的内容了。

2.2.1 IO 总线可以看作网络么

IO 总线可以接入多个外设，比如键盘、鼠标、网卡、显卡、USB 设备、串口设备和并口设备等，最重要的当然要属磁盘设备了。讲到这里，大家的脑海中应该能出现这么一种架构：CPU、内存和各种外设都连接到一个总线上，这不正是以太网 HUB 的模型么？HUB 本身就是一个总线结构而已，所有接口都接在一条总线上，HUB 所做的就是避免总线信号衰减，因此需要电源来加强总线上的电信号。

没错！仔细分析思考提炼之后，确实就是这么一个模型！不过 IO 总线和以太网 HUB 模型还是有些区别。CPU 和内存因为足够快，它们之间单独用一条总线连接。这个总线和慢速 IO 总线之间通过一个桥接芯片连接，也就是主板上的北桥芯片。这个芯片连接了 CPU、内存和 IO 总线。

CPU 与北桥连接的总线叫做系统总线，也称为前端总线。这个总线的传输频率与 CPU 自身频率是两个不同概念，总线频率相当于 CPU 向外部存取数据时的数据传输速率，而 CPU 自身的频率则表示 CPU 运算时电路产生的频率。

> **提示** 本书写作时，Intel 用于 PC 的 CPU 前端总线频率已经可以达到 1600MHz，而作者所使用的 PC，CPU 为 Intel 赛扬Ⅱ，前端总线只有 100MHz，整整 16 倍的提升，而 CPU 自身频率提升不过三四倍而已，但是性能却提升了不止三四倍。

前端总线的条数，比如 64 条或者 128 条，就叫做总线的位数。这个位数与 CPU 内部的位数也是不同的概念，CPU 位数指的是寄存器和运算单元之间总线的条数。内存与北桥连接的总线叫做内存总线。由于北桥速度太快，而 IO 总线速度相对北桥显得太慢，所以北桥和 IO 总线之间，往往要增加一个网桥，叫做南桥，在南桥上一般集成了众多外设的控制器，比如磁盘控制器、USB 控制器等。



这不正是个不折不扣的“网络”么？而且还是个不折不扣的“网桥”！我们看，CPU 和内存是一个冲突域，IO 总线是一个冲突域，桥接芯片将这两个冲突域桥接起来，这正是网桥的思想！太好了！我们的思想在这个模型中得到了升华！我们知道了计算机系统原来就是一个网络啊！

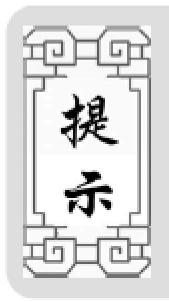
下面就来看看，在这个网络上，我们能够干点什么惊天动地的事呢？



IO 总线其实不是一条总线，它分成数据总线、地址总线和控制总线。寻址用地址总线，发数据用数据总线，发中断信号用控制总线。而且 IO 总线是并行而不是串行的，有 32 位或者 64 位总线。32 位总线也就是说有 32 根导线来传数据，64 位总线用 64 根导线来并行传数据。

2.2.2 CPU、内存和磁盘之间通过网络来通信

CPU 是一个芯片，磁盘是一个有接口的盒子，它们不是一体的而是分开的，而且都连接在这个网桥上。那么 CPU 向磁盘要数据，也就是两个节点之间的通信，必定要通过一种通路来获取，这个通路当然是电路！



当然也可以是辐射的电磁波，估计 21 世纪还应用不到 CPU 上。

凡是分割的节点之间，需要接触和通信，就可以成为网络。那么就不由地使我们往 OSI 模型上去靠，这个模型定义得很好。既然通信是通过电路，也就是物理层的东西，那么链路层都有什么内容呢？

大家知道，链路层相当于一个司机，它把货物运输到对端。司机的作用就是驾驶车辆，而且要判断交通规则做出配合。那么在这个计算机总线组成的网络中，是否也需要这样一个角色呢？答案是不需要。因为各个节点之间的路实在是太短、太稳定了！主板上那些电容、电阻和蛇行线，这一切都是为了保障这些电路的稳定和高速。在这样的一条高速、高成本的道路上，是不需要司机的，更不需要押运员！所以，计算机总线网络是一个只有物理层、网络层和上三层的网络！



所有的网络都可以定义成连起来、找目标和发数据。也就是“连找发”模型，这也是构成一个网络的三元素。任何网络都必须具有这三元素(点对点网络除外)。连，代表物理层。物理层必须要有，如果没有物理层，要达到两点之间通信是不可能的。物理层可以是导线，可以是电磁波，总之必须有物理层。找，突出一个找字，既然要找，那么就要区分方法，也就是编址，比如 IP 等。发，突出一个发字，即指最上层发出数据。

下面我们就按照“连找发”三元素理论，去分析一个 CPU 向磁盘要数据的例子。

CPU 与硬盘数据交互的过程

首先看“连”这个元素，这个当然已经具备了，因为总线已经提供了“连”所需的条件。

再看“找”这个元素，前面说了，首先要有区分，才能有所谓“找”，这个区分体现在主机总线中就是设备地址映射。每个 IO 设备在启动时都要向内存中映射一个或者多个地址，这个地址有 8 位长，又被称作 IO 端口。针对这个地址的数据，统统被北桥芯片重定向到总线上实际的设备上。假如，IDE 磁盘控制器地址被映射到了地址 0xA0，也就是十六进制 A0，CPU 根据程序机器代码，向这个地址发出多条指令来完成一个读操作，这就是“找”。“找”的条件也具备了。

接下来我们看看“发”这个元素！首先 CPU 将这个 IO 地址放到系统总线上，北桥接收到之后，会等待 CPU 发送第一个针对这个外设的指令。然后 CPU 发送如下 3 条指令。

- 第一条指令：指令中包含了表示当前指令是读还是写的位，而且还包含了其他选项，比如操作完成时是否用中断来通知 CPU 处理，是否启用磁盘缓存等。
- 第二条指令：指明应该读取的硬盘逻辑块号(LBA)。这个逻辑块在我们讲磁盘结构时会讲到，总之逻辑块就是对磁盘上存储区域的一种抽象。
- 第三条指令：给出了读取出来的内容应该存放到内存中哪个地址中。

这 3 条指令被北桥依次发送给 IO 总线上的磁盘控制器来执行。磁盘控制器收到第一条指令之后，知道这是读指令，而且知道这个操作的一些选项，比如完成是否发中断，是否启用磁盘缓存等，然后磁盘控制器会继续等待下一条指令，即逻辑块地址(号)。磁盘控制器收到指令之后，会进行磁盘实际扇区和逻辑块的对应查找，可能一个逻辑块会对应多个扇区，查找完成之后，控制器驱动磁头寻道，等盘体旋转到那个扇区后，磁头开始读出数据。在读取数据的同时，磁盘控制器会接收到第三条指令，也就是 CPU 给出的数据应该存放在内存中的地址。有了这个地址，数据读出之后直接通过 DMA 技术，也就是磁盘控制器可以直接对内存寻址并执行写操作，而不必先转到 CPU，然后再从 CPU 存到内存中。数据存到内存中之后，CPU 就从内存中取数据，进行其他运算。

上面说的过程是“读”，“写”的过程也可以以此类推，而且 CPU 向磁盘读写数据，和向内存读写数据大同小异，只不过 CPU 和内存之间有更高速的缓存。缓存对于计算机很重要，对于磁盘阵列同样重要，后面内容将会介绍到。

> 思考
> CPU 在对磁盘发送指令的时候，这些指令是怎么定义的？这些指令其实是发给了主板南桥上集成的(或者是通过 PCI 接入 IO 总线的)控制器，比如 ATA 控制器或者 SCSI 控制器。然后控制器再向磁盘发出一系列的指令，让磁盘读取或者写入某个磁道、某个扇区等。CPU 不需要知道这些，CPU 只需要知道逻辑块地址是读还是写就可以了。让 CPU 产生这些信号的是磁盘控制器驱动程序。

那么控制器对磁盘发出的一系列指令是怎么定义的呢？它们形成了两大体系，一个是 ATA 指令集，一个是 SCSI 指令集。SCSI 指令集比 ATA 指令集高效，所以广泛用于服务器

和磁盘阵列环境中。这些指令集，也可以称为协议，协议就是语言，就是让通信双方知道对方传过来的 bit 流里面到底包含了什么，怎么由笔划组成字，由字组成词，词组成句子，等等。

2.3 网 中 之 网

通过图 2.2，我们可以体会到，计算机的主板上的各个部件本身就形成了一个网络，而且通过网卡，还可以连接到外部网络。

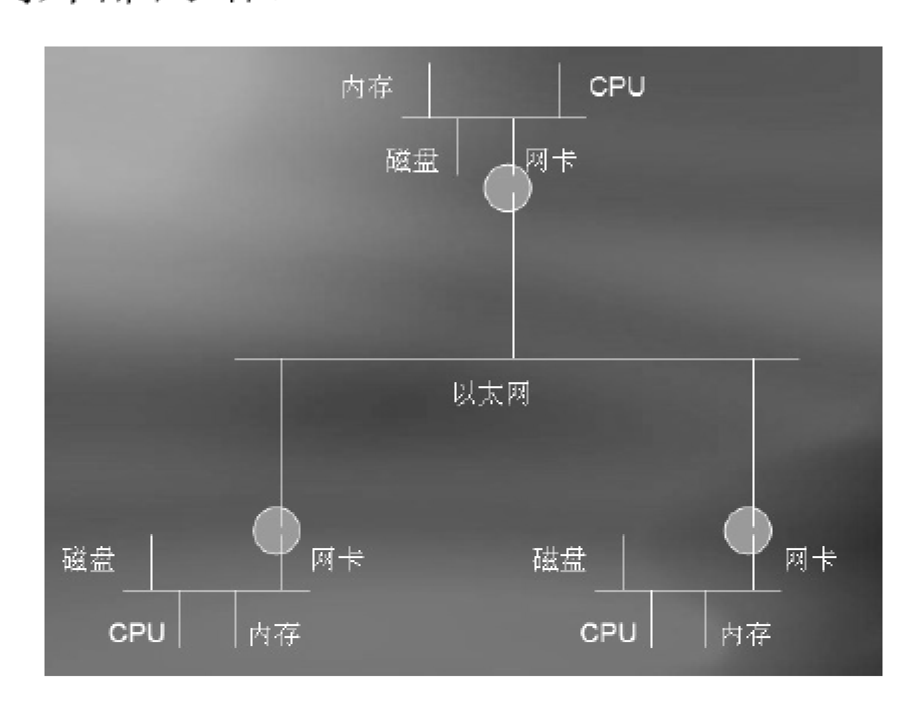


图 2.2 网中之网

正所谓：

CPU 内存和磁盘，
大家都在线上谈。
待当看破三元素，
网中有网天际来！

读 书 笔 记



第三章 磁盘大挪移

磁盘原理与技术详解

- 磁盘结构
- 接口
- 串行
- 并行

磁盘分为软盘和硬盘。将布满磁性粒子的一片圆形软片包裹在一个塑料壳中，中间开孔，以便电机夹住这张软片来旋转，这就是软盘。

将软盘插入驱动器，电机便会带动这张磁片旋转，同时磁头也夹住磁片进行数据读写。软盘和录音带是双胞胎，只不过模样不太一样而已。软盘记录的是数字信号，录音带记录的是模拟信号。软盘上的磁性粒子的磁极，不管是 N 极还是 S 极，其磁化强度都是一样的，磁头只要探测到 N 极，便认为是 1，探测到 S 极，便认为是 0，反过来也可以，这就是用 0 和 1 来记录的数字信号数据。另外，因为软盘被设计为块式的而不是流式的，所以需要进行扇区划分等操作。

所谓块式，就是指数据分成一块块地存放在介质上，可以直接选择读写某一块数据，定位这个块的速度比较快。所谓流式，就是指数据是连续不断地存放在介质上。就像一首歌，不可能让录音机在磁带上定位到这首歌的某处开始播放，只能定位到某首歌曲的前面或者后面。

模拟磁带，也就是录音带，记录是线性连续的，没有扇区的概念，属于流式记录。在每个流之间可以有一段空隙，以便磁头可以通过快进快速定位到这个位置，但是由于设计的原因，磁带定位的速度远比磁盘慢。但是磁带的设计，从一开始就是为了满足大容量数据存储的需要。如果将缠绕紧密的磁带铺展开来，可以想象它的面积比一张磁盘要大得多，所以存储容量必然也就大于磁盘。现在一盘 LTO3 的数字磁带可以在 1 平方分米底面，2 厘米高的体积中存放 400GB 的数据，如果使用压缩技术，可以存放约 800GB 的数据。而它的价格却比同等容量硬盘的一半还低。

但是磁带绝对不可以作为数据实时存储的介质，因为它不可以定位到某个块，这也决定了磁带只能用来做数据备份。Sun 公司的顶级磁带库产品可以达到一台磁带库中存放 1 万盘磁带，最大可以让 32 台磁带库级联，从而形成 32 万盘磁带的大规模磁带库阵列。

而作为本章重点介绍的硬盘技术，不仅存取速度要比软盘更快，随着技术发展带来的成本下降，更有取代磁带机成为普及型数据存储的趋势。

3.1 硬盘结构

1. 结构图

硬盘大致由盘片、读写头、马达、底座、电路板等几大项组合而成，如图 3.1 和图 3.2 所示。



图3.1 磁盘的构成要件

图3.2 磁盘结构图

2. 盘片

盘片的基板由金属或玻璃材质制成，为达到高密度高稳定性的要求，基板要求表面光滑平整不可有任何瑕疵。然后将磁粉溅镀到基板表面上，最后再涂上保护润滑层。此处要应用两项高科技，一是要制造出不含杂质的极细微的磁粉，二是要将磁粉均匀地溅镀上去。

盘片每面粗计密度为 32901120000 bit，可见其密度相当高，所以盘片不可有任何污染，

全程制造均须在 Class 100 高洁净度的无尘室内进行，这也是硬盘要求需在无尘室才能拆解维修的原因。因为磁头是利用气流漂浮在盘片上，并没有接触到盘片，因而可以在各轨间高速来回移动，但如果磁头距离盘片太高读取的讯号就会太弱，太低又会磨到盘片表面，所以盘片表面上必须相当的光滑平整，任何异物和尘埃均会使得磁头摩擦到磁面而造成数据永久性伤害。

3. 磁头

硬盘的储存原理是将数据用其控制电路通过硬盘读写头(Read Write Head)去改变磁盘表面上极细微的磁性粒子簇的 N、S 极性来加以储存，所以这几片磁盘相当重要。

磁盘为了储存更多数据，必须将磁性粒子簇溅镀在磁头可定位的范围内，并且磁性粒子制作得越小越好。经过溅镀，磁盘表面上磁粒子密度相当的高，而硬盘读写头为了能在磁盘表面高速来回移动读取数据则需漂浮在磁盘表面上，但是不能接触，接触就会造成划伤。磁头如果太高的话读取到的信号就会很弱，无法达到高稳定性的要求，所以要尽可能压低，其飞行高度(Flying Height)非常小(可比喻成要求一架波音 747 客机，其飞行高度须保持在 1 米的距离而不可坠毁)。实现这种技术，完全是靠磁盘旋转时，在盘片上空产生气流，利用空气动力学使磁头悬浮于磁片上空。磁头厚度如图 3.3 所示。

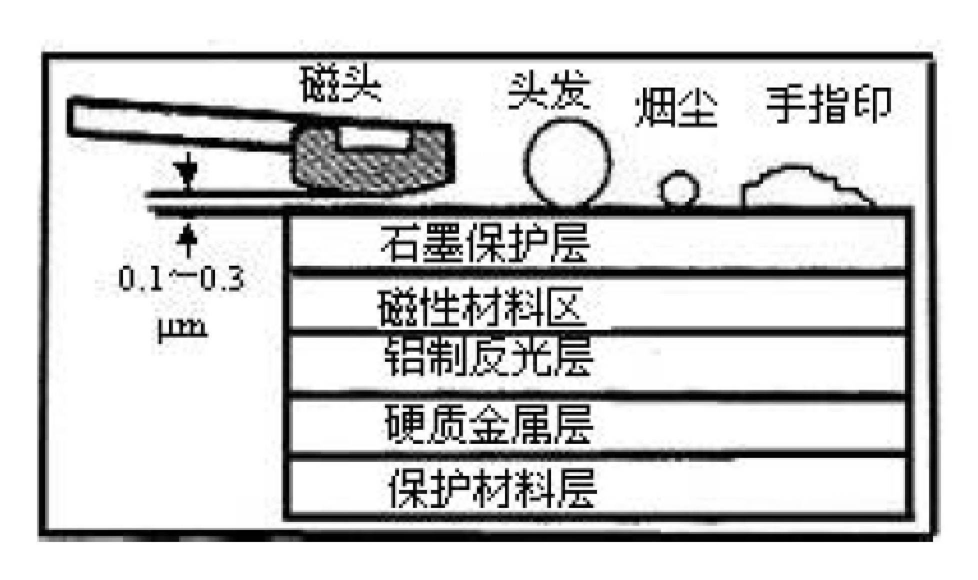


图3.3 磁头厚度示意图

早期的硬盘在每次关机之前需要运行一个被称为 Parking 的程序，其作用是让磁头回到盘片最内圈的一个不含磁粒子的区域，叫做启停区。硬盘不工作时，磁头停留在启停区，当需要从硬盘读写数据时，磁盘就先开始旋转。旋转速度达到额定速度时，磁头就会因盘片旋转产生的气流抬起来，这时磁头才向盘片中存放数据的区域移动。盘片旋转产生的气流相当强，足以托起磁头，并与盘面保持一个微小的距离。这个距离越小，磁头读写数据的灵敏度就越高，当然对硬盘各部件的要求也就越高。

早期设计的磁盘驱动器可使磁头保持在盘面上方几微米处飞行，稍后的一些设计使磁头在盘面上的飞行高度降到约 0.1μm～0.5μm，现在的水平已经达到 0.005μm～0.01μm，只是人类头发直径的千分之一。气流既能使磁头脱离开盘面，又能使它保持在离盘面足够近的地方，非常紧密地随着磁盘表面呈起伏运动，使磁头飞行处于严格受控状态。磁头必须飞行在盘面上方，而不接触盘面，这种距离可避免擦伤磁性涂层，而更重要的是不让磁性涂层损伤磁头。但是，磁头也不能离盘面太远，否则就不能使盘面达到足够强的磁化，难以读出盘上的数据。



硬盘驱动器磁头的飞行悬浮高度低、速度快，一旦有小的尘埃进入硬盘密封腔内或者磁头与盘体发生碰撞，就有可能造成数据丢失形成坏块，甚至造成磁头和盘体的损坏。所以，硬盘系统的密封一定要可靠，在非专业条件下绝对不能开启硬盘密封腔，否则灰尘进入后会加速硬盘的损坏。另外，硬盘驱动器磁头的寻道伺服电机多采用音圈式旋转或直线运动步进电机，在伺服跟踪的调节下精确地跟踪盘片的磁道，所以硬盘工作时不要有冲击碰撞，搬动时也要小心轻放。

4. 步进电机

为了让磁头精确定位到每个磁道，用普通的电机达不到这样的精度，必须用步进电机，利用精确的齿轮组或者音圈，每次旋转可以仅仅使磁头进行微米级的位移。音圈电机则是使用精密缠绕的铜丝，置于磁场之中，通过控制电流的流向和强度，使得磁头臂在磁场作用下做精确的步进。

3.1.1 盘片上的数据组织

硬盘上的数据是如何组织与管理的呢？硬盘首先在逻辑上被划分为磁道、柱面以及扇区，其结构关系如图 3.4 所示。

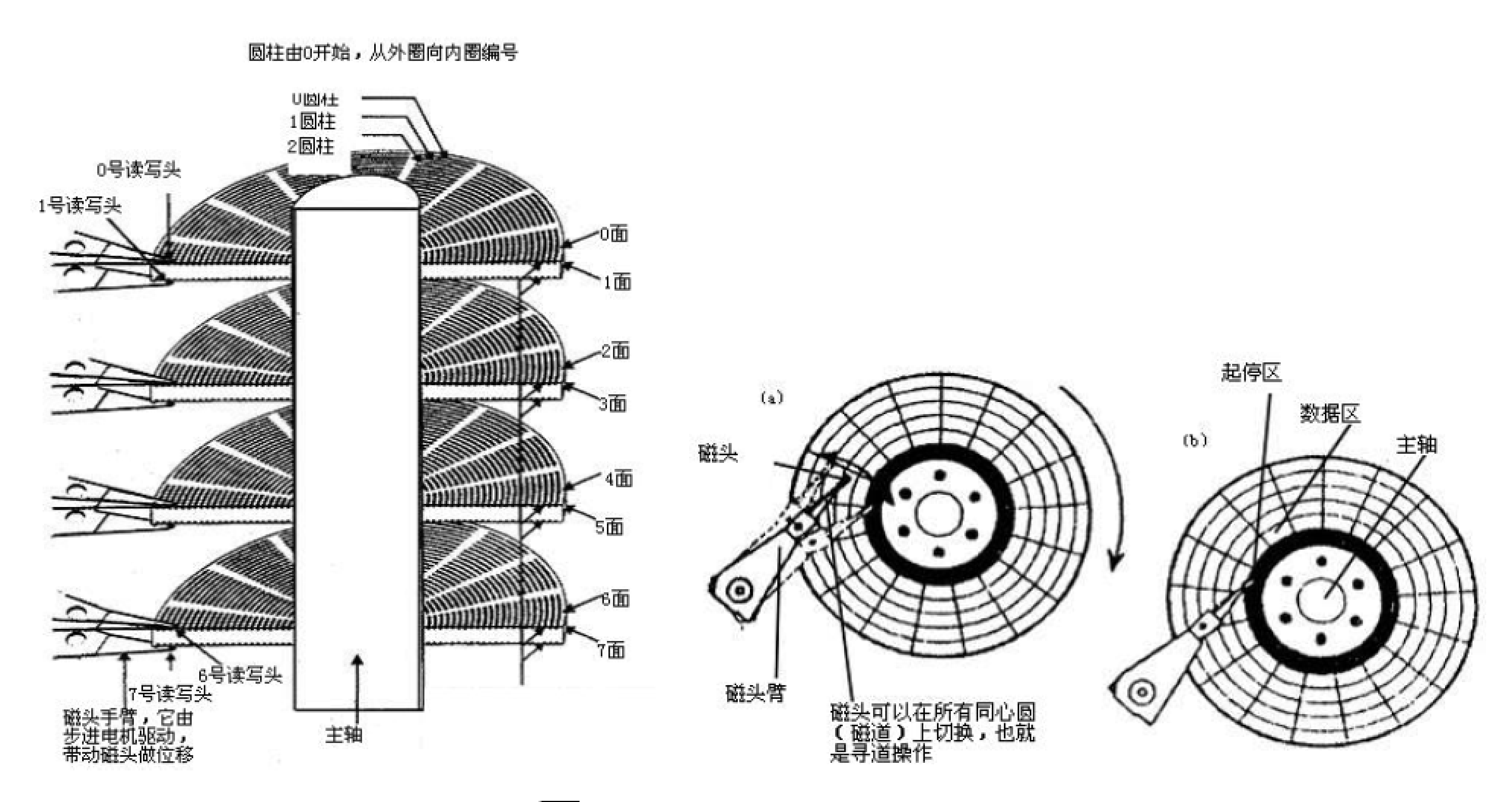


图3.4　柱面和盘片上的磁道

每个盘片的每个面都有一个读写磁头，磁头起初停在盘片的最内圈，即线速度最小的地方。这是一个特殊区域，它不存放任何数据，称为启停区或着陆区(Landing Zone)。启停区外就是数据区。在最外圈，离主轴最远的地方是 0 磁道，硬盘数据的存放就是从最外圈开始的。

那么，磁头是如何找到 0 磁道的位置呢？从图 3.4 中可以看到，有一个 0 磁道检测器，由它来完成硬盘的初始定位。0 磁道为存放着用于操作系统启动所必需的程序代码，因为 PC 启动后 BIOS 程序在加载任何操作系统或其他程序时，总是默认从磁盘的 0 磁道读取程

序代码来运行。



0 磁道是如此重要，以致于很多硬盘仅仅因为 0 磁道损坏就报废了，这是非常可惜的。

下面对盘面、磁道、柱面和扇区的含义逐一进行介绍。

1. 盘面

硬盘的盘片一般用铝合金材料做基片，高速硬盘也有用玻璃做基片的。玻璃基片更容易达到所需的平面度和光洁度，而且有很高的硬度。磁头传动装置是使磁头作径向移动的部件，通常有两种类型的传动装置，一种是齿条传动的步进电机传动装置，另一种是音圈电机传动装置。前者是固定推算的传动定位器，而后者则采用伺服反馈返回到正确的位置上。磁头传动装置以很小的等距离使磁头部件做径向移动，用以变换磁道。

硬盘的每一个盘片都有两个盘面，即上、下盘面。每个盘面都能利用，都可以存储数据，成为有效盘片。每一个这样的有效盘面都有一个盘面号，按从上到下的顺序从 0 开始依次编号。在硬盘系统中，盘面号又叫磁头号，因为每一个有效盘面都有一个对应的读写磁头。硬盘的盘片组在 2～14 片不等，通常有 2～3 个盘片，故盘面号(磁头号)为 0～3 或 0～5。

2. 磁道

磁盘在格式化时被划分成许多同心圆，这些同心圆轨迹叫做磁道。磁道从最外圈向内圈从 0 开始顺序编号。硬盘的每一个盘面有 300～1024 个磁道，新式大容量硬盘每面的磁道数更多。这些同心圆磁道不是连续记录数据，而是被划分成一段段的圆弧，这些圆弧的角速度一样。由于径向长度不一样，所以线速度也不一样，外圈的线速度较内圈的线速度大。在同样的转速下，外圈在同样时间段里，划过的圆弧长度要比内圈划过的圆弧长度大，因此外圈数据的读写要比内圈快。

每段圆弧叫做一个扇区，扇区从 1 开始编号，每个扇区中的数据作为一个单元同时读出或写入，是读写的最小单位。不可能发生读写半个或者四分之一个这种小于一个扇区的情况，因为磁头只能定位到某个扇区的开头或者结尾，而不能在扇区内部做定位。所以，一个扇区内部的数据，是连续流式记录的。一个标准的 3.5 英寸硬盘盘面通常有几百到几千条磁道。磁道是肉眼看不见的，只是盘面上以特殊形式磁化了的一些磁化区。划分磁道和扇区的过程，叫做低级格式化，通常在硬盘出厂的时候就已经格式化完毕了。相对于低级格式化来说，高级格式化指的是对磁盘上所存储的数据进行文件系统的标记，而不是对扇区和磁道进行磁化标记。

3. 柱面

所有盘面上的同一磁道，在竖直方向上构成一个圆柱，通常称作柱面。每个圆柱上的磁头由上而下从 0 开始编号。数据的读写按柱面进行，即磁头读写数据时首先在同一柱面内从 0 磁头开始进行操作，依次向下在同一柱面的不同盘面(即磁头)上进行操作。只有在同一柱面所有的磁头全部读写完毕后磁头才转移到下一柱面，因为选取磁头只需通过电子切

换即可，而选取柱面则必须通过机械切换，即寻道。

电子切换相当快，比使用机械将磁头向邻近磁道移动要快得多，所以数据的读写按柱面进行，而不按盘面进行。也就是说，一个磁道写满数据后，就在同一柱面的下一个盘面来写。一个柱面写满后，才移到下一个柱面开始写数据，这样可以减少寻道的频繁度。读写数据也按照这种方式进行，这样就提高了硬盘的读写效率。

一块硬盘驱动器的圆柱数或每个盘面的磁道数既取决于每条磁道的宽窄(也与磁头的大小有关)，也取决于定位机构所决定的磁道间步距的大小。如果能将磁头做得足够精细，定位距离足够小，那么就会获得更高的磁道数和存储容量。如果磁头太大，则磁道数就要降低以容纳这个磁头，这样磁道与磁道之间的磁粉将无法利用，浪费得太多。如果能将磁头做成单个原子的精度，那么存储技术就会发生革命性的质变。



利用原子探针来移动物质表面的原子成特定形状，这种技术早已实现。如果能将这种技术应用到数据存储领域，则存储容量和速度将会以几何倍数上升。

4. 扇区

1) 扇区头标

将每个环形磁道等距离切割，形成等长度的圆弧，每个圆弧就是一个扇区。划分扇区的目的是为了使数据存储更加条理化，就像一个大仓库要划分更多的房间一样。每个扇区可以存放 512 个字节的数据和一些其他信息。一个扇区有两个主要部分：存储数据地点的标识符和存储数据的数据段，如图 3.5 所示。

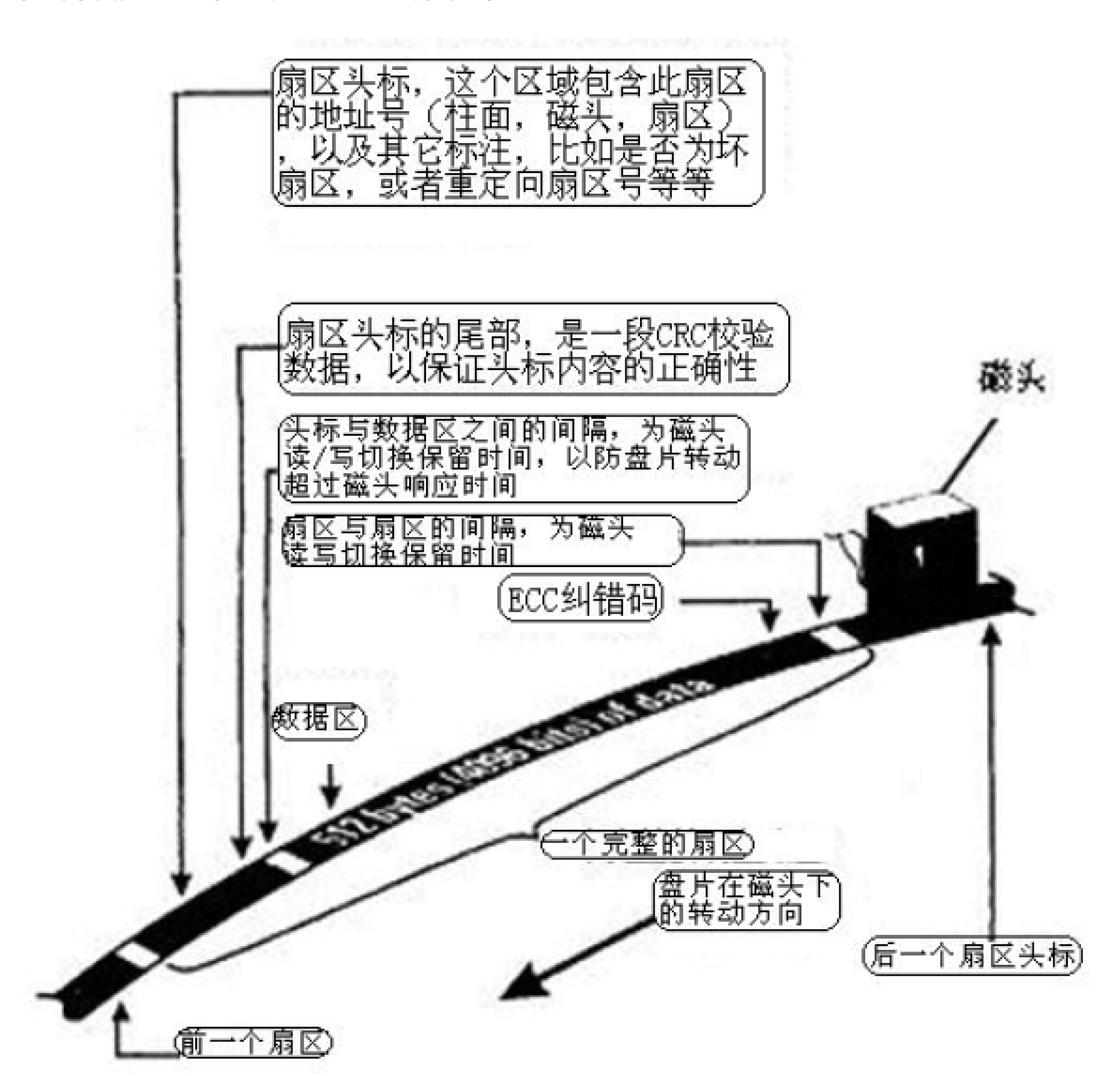


图3.5 扇区示意图

扇区头标包括组成扇区三级地址的三个数字。

- 扇区所在的柱面(磁道)。
- 磁头编号。

- 扇区在磁道上的位置，即扇区号。

柱面(Cylinder)，磁头(Header)和扇区(Sector)三者简称 CHS，所以扇区的地址又称为CHS 地址。

磁头通过读取当前扇区的头标中的 CHS 地址，就可以知道当前是处于盘片上的哪个位置，比如是内圈还是外圈，哪个磁头正在读写(同一时刻，只能有一个磁头在读写)等。

CHS 编址方式在早期的小容量硬盘中非常流行，但是目前的大容量硬盘的设计和低级格式化方式已经有所变化，所以 CHS 编址方式已经不再使用，而转为 LBA 编址方式。LBA 编程方式不再划分柱面和磁头号，这些数据由硬盘自身保留，而磁盘对外提供的地址全部为线性的地址，即 LBA 地址。

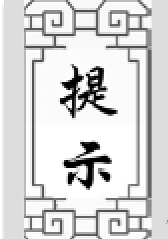
所谓线性，指的是把磁盘想象成只有一个磁道，这个磁道是无限长的直线，扇区为这条直线上的等长线段，从 1 开始顺序编号，直到无限远。显然，这种方式屏蔽了柱面、磁头这些复杂的东西，向外提供了简单的方式，所以非常利于编程。然而磁盘中的控制电路依然要找到某个 LBA 地址到底对应着哪个磁道哪个磁头上的哪个扇区，这种对应关系保存在磁盘控制电路的 ROM 芯片中，磁盘初始化的时候载入缓存中以便随时查询。



基于 CHS 编址方式的磁盘最大容量

- 磁头数(Heads)表示硬盘总共有几个磁头，也就是有几面盘片，最大为 255 (用 8 个二进制位存储)。
- 柱面数(Cylinders) 表示硬盘每一面盘片上有多少条磁道，最大为 1023(用 10 个二进制位存储)。扇区数(Sectors) 表示每一条磁道上有多少扇区，最大为 63(用 6 个二进制位存储)。
- 每个扇区一般是 512 个字节，理论上讲这不是必须的。目前很多大型磁盘阵列所使用的硬盘，由于阵列控制器需要做一些诸如校验信息之类的特殊存储，这些磁盘都被格式化为每扇区 520 字节。

如果按照每扇区 512 字节来计算，磁盘最大容量为 255×1023×63×512/1048576 = 8024 GB (1MB =1048576 Bytes)。这就是所谓的 8GB 容量限制的原因。但是随着技术的不断发展，CHS 地址的位数在不断增加，所以可寻址容量也在不断增加。



磁盘驱动器内怎样放下 255 个磁头呢？这是不可能的。目前的硬盘一般可以有 1 盘片、2 盘片或者 4 盘片，这样就对应着 2、4 磁头或者 8 磁头。那么这样算来，硬盘实际容量一定小于 8GB 了？显然不是这样的。所谓 255 个磁头，这只是一个逻辑上的说法，实际的磁头、磁道、扇区等信息都保存在硬盘控制电路的 ROM 芯片中。而每条磁道上真的最多只有 64 个扇区么？当然也不是，一条磁道上实际的扇区数远远大于 64，这样就分摊了磁头数实际少于 255 个所产生的“容量减小”。所以，只是 CHS 编址方式沿袭了老的传统，不愿意去做修改导致的。而这种沿袭达到了一定容忍程度之后，最终导致 LBA 编址方式替代了 CHS 编址方式。

头标中还包括一个字段，其中有显示扇区是否能可靠存储数据，或者是已发现某个故障因而不宜使用的标记。有些硬盘控制器在扇区头标中还记录有指示字，可在原扇区出错时指引磁头跳转到替换扇区或磁道。最后，扇区头标以循环冗余校验 CRC 值作为结束，以供控制器检验扇区头标的读出情况，确保准确无误。

2) 扇区编号和交叉因子

给扇区编号的最简单方法是 l、2、3、4、5、6 等顺序编号。如果扇区按顺序绕着磁道依次编号，那么磁盘控制电路在处理一个扇区的数据期间，可能会因为磁盘旋转太快，没等磁头反应过来，已经超过扇区间的间隔而进入了下一个扇区的头标部分，则此时磁头若想读取这个扇区的记录，就要再等一圈，等到盘片旋转回来之后再次读写，这个等待时间无疑是非常浪费的。

显然，要解决这个问题，靠加大扇区间的间隔是不现实的，那会浪费许多磁盘空间。许多年前，IBM 的一位杰出工程师想出了一个绝妙的办法，即对扇区不使用顺序编号，而是使用一个交叉因子(Interleave)进行编号。交叉因子用比值的方法来表示，如 3:1 表示磁道上的第 1 个扇区为 1 号扇区，跳过两个扇区即第 4 个扇区为 2 号扇区，这个过程持续下去直到给每个物理扇区编上逻辑号为止。

例如，每磁道有 17 个扇区的磁盘按 2:1 的交叉因子编号就是 l，10，2，11，3，12，4，13，5，14，6，15，7，16，8，17，9。而按 3:1 的交叉因子编号就是 l，7，13，2，8，14，3，9，15，4，10，16，5，11，17，6，12。当设置 1:1 的交叉因子时，如果硬盘控制器处理信息足够快，那么读出磁道上的全部扇区只需要旋转一周。但如果硬盘控制器的处理动作没有这么快，则只有磁盘所转的圈数等于针对这个磁道的交叉因子时，才能读出每个磁道上的全部数据。将交叉因子设定为 2:1 时，磁头要读出磁道上的全部数据，磁盘只需转两周。如果 2:1 的交叉因子仍不够慢，这时可将交叉因子调整为 3:1，如图 3.6 所示。

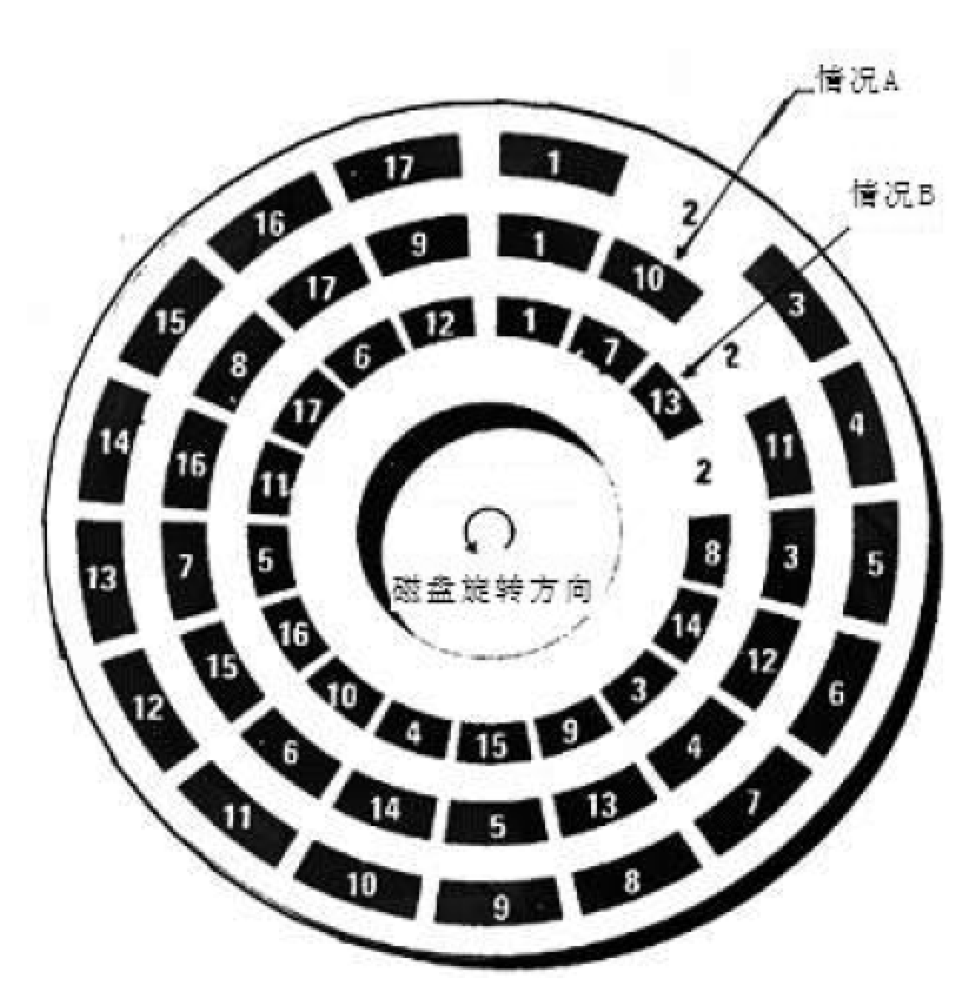


图3.6 MFM 改进型交叉因子示意图

图 3.6 所示是典型的 MFM(Modified Frequency Modulation，改进型调频制编码)硬盘，每磁道有 17 个扇区，画出了用三种不同的扇区交叉因子编号的情况。最外圈的磁道(0 号柱面)上的扇区用简单的顺序连续编号，相当于扇区交叉因子是 1:1。1 号磁道(柱面)的扇区按

2:1 的交叉因子编号，而 2 号磁道按 3:1 的扇区交叉因子编号。

在早期的硬盘管理工作中，设置交叉因子需要用户自己完成。用 BIOS 中的低级格式化程序对硬盘进行低级格式化时，就需要指定交叉因子，有时还需要设置几种不同的值来比较其性能，而后确定一个比较好的值。现在的硬盘 BIOS 已经自己解决了这个问题，所以一般低级格式化程序中就不再提供这一设置选项了。

系统将文件存储到磁盘上时，是按柱面、磁头、扇区方式进行的，即最先是第 1 磁道的第 1 磁头下(也就是第 1 盘面的第一磁道)所有的扇区，然后是同一柱面的下一磁头，直到整个柱面都存满。系统也是以相同的顺序去读出数据。读数据时通过告诉磁盘控制器要读出数据所在的柱面号、磁头号和扇区号(物理地址的三个组成部分)进行读取(现在都是直接使用 LBA 地址来告诉磁盘所要读写的扇区)。磁盘控制电路则直接将磁头部件步进到相应的柱面，选中相应磁头，然后立即读取当前磁头下所有的扇区头标地址，然后把这些头标中的地址信息与期待检出的磁头和柱面号做比较。如果不是要读写的扇区号则读取扇区头标地址进行比较，直到相同以后，控制电路知道当前磁头下的扇区就是要读写的扇区，然后立即让磁头读写数据。

如果是读数据，控制电路会计算此数据的 ECC 码，然后把 ECC 码与已记录的 ECC 码相比较；如果是写数据，控制电路会计算出此数据的 ECC 码，存储到数据部分的末尾。在控制电路对此扇区中的数据进行必要的处理期间，磁盘会继续旋转。由于对信息的后处理需要耗费一定的时间，在这段时间内磁盘可能已旋转了相当的角度。

交叉因子的确定是一个系统级的问题。一个特定的硬盘驱动器的交叉因子取决于磁盘控制器的速度、主板的时钟速度、与控制电路相连的输出总线的操作速度等。如果磁盘的交叉因子值太高，就需多花一些时间等待数据在磁盘上存入和读出。相反，太低也同样会影响性能。

前面已经说过，系统在磁盘上写入信息时，写满一个磁道后会转到同一柱面的下一个磁头，当柱面写满时，再转向下一柱面。从同一盘面的一个磁道转到另一个磁道，也就是从一个柱面转到下一个柱面，这个动作叫做换道。在换道期间磁盘始终保持旋转，这就会带来一个问题：假定系统刚刚结束了对一个磁道前一个扇区的写入，并且已经设置了最佳交叉因子比值，现在准备在下一磁道的第一扇区写入，这时必须等到磁头换道结束，让磁头部件重新定位在下一道上。如果这种操作占用的时间超过了一点，尽管是交叉存取，磁头仍会延迟到达。这个问题的解决办法是以原先磁道所在位置为基准，把新的磁道上全部扇区号移动约一个或几个扇区位置，这就是磁头扭斜。磁头扭斜可以理解为柱面与柱面之间的交叉因子，已经由生产厂家设置好，一般不用去改变它。磁头扭斜的更改比较困难，但是它们只在文件很长、超过磁道结尾进行读出和写入时才发挥作用，所以扭斜设置不正确所带来的时间损失比采用不正确的扇区交叉因子值带来的损失要小得多。交叉因子和磁头扭斜可用专用工具软件来测试和更改，更具体的内容这里就不再详述了，毕竟现在很多用户都没有见过这些参数。

提示 最初，硬盘低级格式化程序只是行使有关磁盘控制器的专门职能来完成设置任务。由于这个过程可能会破坏低级格式化的磁道上的全部数据，现在也极少采用了。

扇区号存储在扇区头标中，扇区交叉因子和磁头扭斜的信息也存放在这里。

扇区交叉因子由写入到扇区头标中的数字设定，所以，每个磁道可以有自己的交叉因子。在大多数驱动器中，所有磁道都有相同的交叉因子。但有时因为操作上的原因，也可能导致各磁道有不同的扇区交叉因子。比如在交叉因子重置程序工作时，由于断电或人为中断就会造成一些磁道的交叉因子发生了改变，而另一些磁道的交叉因子没有改变。这种不一致性对计算机不会产生不利影响，只是有最佳交叉因子的磁道要比其他磁道的工作速度更快。

3.1.2 硬盘控制电路简介

我们在了解了磁盘的结构之后，知道了磁盘是靠磁性子来存放数据的，那么有人就会问了：一个磁性子到底是什么概念？是一个磁性分子么？不是，这个“子”的概念是指一个区域，这个区域存在若干磁性分子，这些分子聚集到一起，直到磁头可以感觉到它的磁性为止。所以和磁带一样，磁记录追根到底就是利用线性中的段。根据这一段区域上的一片分子是 N 极还是 S 极，然后将其转换成电信号，也就产生了字节，从而记录了数据。当然只有存储介质还远远不够，要让数据可以被读出，被写入，还要有足够的速度和稳定性满足人们的需求，这就需要配套的电路了。

图 3.7 给出一个完整详细的硬盘电路示意框图。

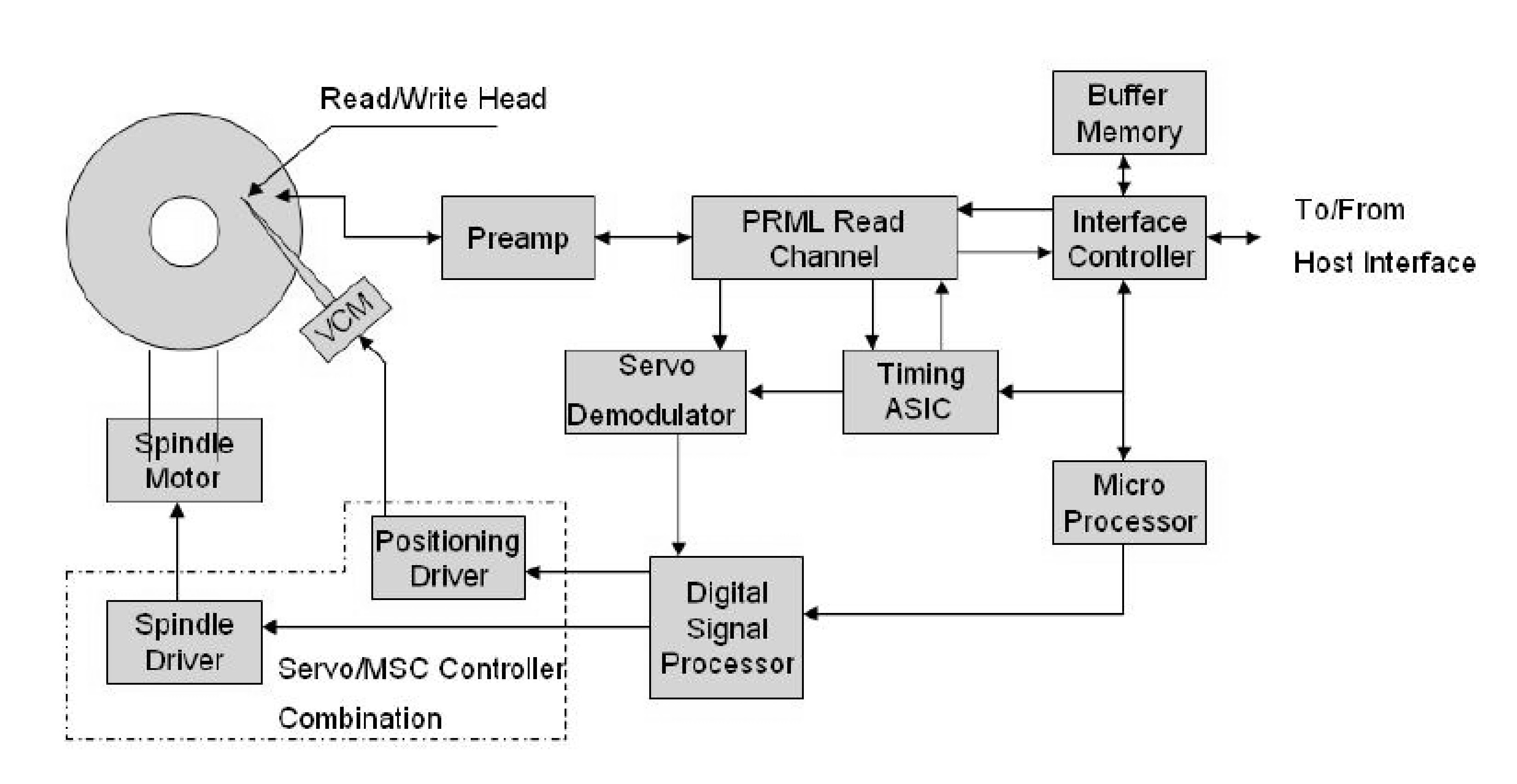


图3.7 硬盘控制电路示意图

硬盘电路由 14 个部分组成。

- Buffer Memory：缓冲区存储器。
- Interface Controller：接口控制器。
- Micro-Controller：微控制器，缩写为 MCU。
- PRML Partial-Response Maximum-Likelihood Read Channel。
- Timing ASIC：时间控制专用集成电路。
- Servo Demodulator：伺服解调器。
- Digital Signal Processor(DSP)：数字信号处理器。
- Preamp：预放大器。
- Positioning Driver：定位驱动器。

- VCM(Voice Coil Motor)：音圈电动机。
- Magnetic Media Disk：磁介质盘片。
- SpindleMotor：主轴电机。
- Spindle Driver：主轴驱动器。
- Read/Write Head：读/写磁头。

实际电路不会有这么多一片一片的独立芯片，硬盘生产厂家在设计电路时都是选取高度集成的 IC 芯片，这样既减小了体积又提高了可靠性。当然这也正是芯片厂商努力的目标。

大家可以看到图 3.7 中的 Spindle Driver 与 Positioning Driver 这两部分用虚线圈了起来，并且标注了 Servo/MSC Controller Combination 字样。其中 MSC 是 Motor Speed Control 的缩写，意思是伺服/电机速度控制器组合。我们现在能看到的硬盘电路板中就有这么一块合并芯片。

3.1.3 磁盘的 IO 单位

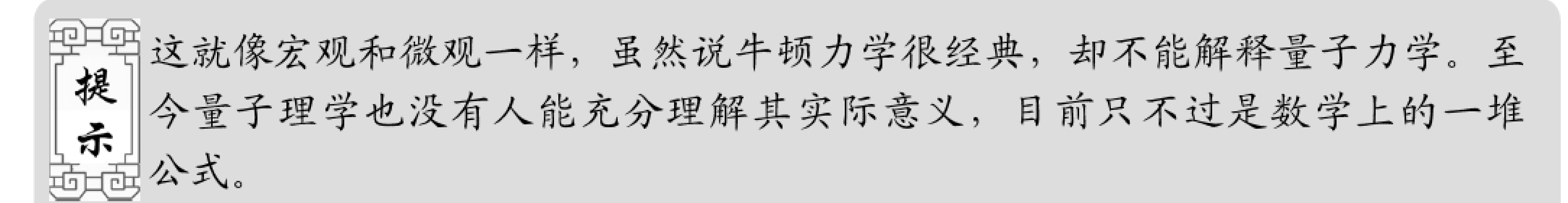
到此，大家应该对磁盘的构造有所理解了。磁盘读写的时候都是以扇区为最小寻址单位的，也就是说不可能往某某扇区的前半部分写入某某数据。一个扇区的大小是 512 个字节，每次磁头连续读写的时候，只能以扇区为单位，即使一次只写了一个字节的数据，那么下一次就不能再向这个扇区剩余的部分接着写入，而是要寻找一个空扇区来写。

对于磁盘来说，一次磁头的连续读或者写叫做一次 IO。



这里的措辞：“对于磁盘来说”。

IO 这个概念，充分理解就是输入输出。我们知道从最上层到最下层，层次之间存在着太多的接口，这些接口之间每次交互都可以称作一次 IO，也就是广义上的 IO。比如卷管理程序对磁盘控制器驱动程序 API 所作的 IO，一次这种 IO 可能要产生针对磁盘的 N 个 IO，也就是说上层的 IO 是稀疏的、简单地，越往下层走越密集、越复杂。



提示 这就像宏观和微观一样，虽然说牛顿力学很经典，却不能解释量子力学。至今量子理学也没有人能充分理解其实际意义，目前只不过是数学上的一堆公式。

除了卷管理程序之外，凌驾于卷管理之上的文件系统对卷的 IO，就比卷更稀疏简单了。同样，上层应用对文件系统 API 的 IO 是更加简单，只需几句代码、几个调用就可以了。比如 Open()某个文件，Seek()到某个位置，Write()一段数据，Close ()这个文件等，就是一次 IO。而就是这一次 IO，可能对应文件系统到卷的 N 个 IO，对应卷到控制器驱动的 N 乘 N 个 IO，对应控制器对最终磁盘的 N×N×N 个 IO。总之，磁盘一次 IO 就是磁头的一次连续读或者写。而一次连续读或者写的过程，不管读写了几个扇区，扇区剩余部分均不能再使用。这无疑是比较浪费的，但是没有办法，总得有个最小单位。

关于最小单位——龟兔赛跑悖论

龟在兔子前面 100 米，兔子的速度是龟的 10 倍。龟对兔子说：“我们同时起跑，你沿直线追我，你永远也追不上我”。这个结论猛一看，会觉得荒唐至极！可是龟分析了：“兔子跑到 100 米我当前的位置时，我同时也向前跑了 10 米。然后兔子跑了 10 米的时候，而我同时也向前跑了 1 米。它再追 1 米，而我又跑了 0.1 米。依此类推，兔子永远追不上我。大家看到这里就糊涂了，这么一分析确实是追不上，但实事却是能追上。那么问题出在哪里呢？

我们分析一下，假如兔子速度是每秒 100 米，龟速度每秒 10 米。首先兔子追出 100 米时候，用时 1 秒，此时龟在兔子前方 10 米处。然后兔子再追出 10 米，用时 0.1 秒，此时龟在前方 1 米处。接着兔子再追出 1 米，用时 0.01 秒，学过小学算术的都能算出来，兔子掉入了一个无限循环小数中，什么时候结束了循环，才能追上龟。那么这就悖论了，小数是无限循环的，这到底是多少秒呢？如果时间可以以无限小的单位延伸，那么兔子确实永远也追不上龟。虽然时间确实是连续的，时间没有理由不连续，时间是一个思想中的概念，时间不是物质，所以时间是惟一能连续的东西，既然时间是无限的、连续的，那么兔子按理说追不上龟了，但是实事确实能追上，但有一个元素我们忽略了，它就是长度的最小单位！！！仔细分析一下，时间和长度是对应的，时间可以无限小，那么这个无限小的时间也应该对应无限小的长度，这样悖论到这里就解决了！因为存在一个长度的最小单位，而没有无限小！也就是说，当兔子走的长度是最小长度时龟就黔驴技穷了，因为不可能再行走比这长度更小的距离了，那么兔子自然就超过了龟。而这个时间是很短暂的，它发生在有限个 1 的时间点上。至于这个最小长度，据说有人计算出来了，它可能是一个原子的长度，也可能比这还小。目前看来，我们移动的时候，最小似乎也不可能移动半个原子的距离！

芝诺悖论(龟兔赛跑悖论)证明了，对于我们目前可观察到的世界来说，是有一个最小距离单位的。如果我们以这个结论为前提，就可以推翻芝诺悖论了。假设这个距离最小单位是一块石头的长度。开始，兔子在乌龟后面相隔 2 块石头的距离，同样兔子的速度是乌龟的 2 倍，按照量子距离理论，这个 2 倍速度，不是无限可分的，那么我们表达这个 2 倍速的时候，应该这么说：兔子每前进 2 块石头的时间，乌龟只能前进一块石头的距离，而不可能前进半块石头。这样的前提下，连小学生都可以计算兔子何时追赶上乌龟了。

3.2 磁盘的通俗演绎

想象一张很大很大的白纸，你要在上面写日记。当你写满这张白纸之后，如果某天想查看某条日记，即无疑将是个噩梦，因为白纸上没有任何格子或行分割线等，你只能通过一行一行地读取日记，搜索你要查看的那条记录。如果给白纸打上格子或行分割线，那么不但书写起来不会凌乱，而且还工整。

那么对于一张上面布满磁性介质的盘片来说，想要在它上面记录数据，如果不给它打格子划线的话，无疑就无法达到块级的记录。所以在使用之前，需要将其低级格式化，也就是划分扇区(格子)。我们见过稿纸，上面的格子是方形阵列排布的，原因很简单，因为稿

纸是方形的。那么对于圆形来说，格子应该怎么排布呢？答案是同心圆排布，一个同心圆(磁道)，就类似于稿纸上的一行，而这一行之内又可以排列上很多格子(扇区)。每个盘片上的行密度、每行中的格子(扇区)密度都有标准来规定，就像稿纸一样。

1】 我们把稿纸放入打印机。

1】 打印机的打印头按照格子的距离精确地做着位移，并不停地喷出墨水，将字体打入纸张上的格子里。

2】 一旦一行打满，走纸轮精确地将稿纸位移到下一行，打印头在这一行上水平位移打满格子。

3】 走纸轮竖直方向位移，打印头水平方向位移，形成了方形扫描阵列，能够定位到整张纸上的每个格子。

同样，我们把圆形盘片安装到一个电机(走纸轮)上，然后在盘片上方加一个磁头(打印喷头)。但是和打印机不同的是，做换行这个动作不是由走纸轮来完成，而是由磁头来完成，称作径(半径)向扫描，也就是在不同同心圆上做切换(换行)。同样做行内扫描这个动作是由电机(走纸轮)而不是磁头(打印头)来完成，称作线扫描(沿着同心圆的圆周进行扫描)。

形成这种角色倒置的原因，很显然是由圆形的特殊性决定的。做圆周运动毕竟比做水平竖直运动要复杂，如果让磁头沿着同心圆做线扫描，则需要将磁头放在一个可以旋转的部件上，此时磁头动而盘片不动，可以达到相同的目的，但是技术难度就复杂多了。因为磁头上有电路连接着磁头和芯片。如果让磁头高速旋转，磁头动而芯片不动，电路的连通性怎么保证？不如让盘片转动来得干脆利索。

和打印机一样，定位到某个特定的格子之后，磁头开始用磁性来对这个格子中的每个磁粒子区做磁化操作，每个磁极表示一个 0 或者 1 状态。每个格子规定可以存放 4096 位这种状态，也就是 512 个字节(很多供大型机使用的磁盘阵列上的磁盘是用 520 字节为一个扇区)。这就像打印机在一个格子再次细分，形成 24×24 点阵，每个坐标上的一个点都对应一种色彩。只不过对于磁盘来说只有 0 或者 1，而对于打印机来说，可以是各种色彩中的一种(黑白打印机也只有黑或者白两种状态)。

磁盘的扇区中没有点阵，一个扇区可以把它看作是线性的。它没有宽，只有长，记录是顺序的，不能像打印机那样可以定位到扇区中的某个点。然而，磁盘比打印机有先天的优势。打印机只能从头到尾打印，而且打印之后不能更改。磁盘却可以对任意的格子进行写入、读取和更改等操作。打印机的走纸轮和打印喷头移动起来很慢，而且嘎嘎作响，听了都费劲。而磁盘的转速则快很多，目前可以达到每分钟 15000 转。磁头的位移动作也非常快，它使用步进电机来精确地换行(换磁道)。但是相对于盘片的转动而言，步进的速度就慢多了，所以制约磁盘性能的主要因素就是这个步进速度(换行或者换道速度)，也就是寻道速度。

如果从最内同心圆换到最外同心圆，耗费的时间无疑是最长的。目前磁盘的平均寻道速度最高可以达到 5ms 多，不同磁盘寻道速度不同，普通 IDE 磁盘可能会超过 10ms。有了这个磁盘记录模型，我们就该研究怎么将这个模型抽象虚拟化出来，让向磁盘写数据的人感觉使用起来非常方便。就像打印机一样，点一下打印，一会纸就蹭蹭地往外冒。下面还是要一层一层地来做，不能直接就抽象到这么高的层次。

首先，要精确寻址每个格子就一定需要给每个格子一个地址。早期的磁盘都是用“盘片，磁道，扇区”来寻址的，一个磁盘盒子中可能不止一片盘片，就像一叠稿纸中有好几

张纸一样。一个盘片上的某一“行”也就是某个磁道，应该可以再区分。一个磁道上的某个扇区也可以区分。到这，就是最终可寻址的最小单位了，而不能再精确定位到一个扇区中的某个点了。磁头只能顺序地写入或者读取出这些点，而不能只更新或者读取其中某个点。也就是说磁头只能一次成批写入或者读取出一个扇区的内容，而不能读写半个或者四分之一个扇区的内容。

后来的扇区寻址体系变了，因为后来的磁盘中每个磁道的扇区数目不同了，外圈由于周长比较长，所以容纳的扇区可以很多，干脆采用了逻辑地址来对每个扇区编址，将具体的盘片、磁道和扇区，抽象成 LBA(logical block address，顺序编址)。LBA1 表示 0 号盘片 0 号磁道的 0 号扇区。依此类推，LBA 地址到实际的盘片、磁道和扇区地址的映射工作由磁盘内部的逻辑电路来查询 ROM 中的对应表而得到，这样就完成了物理地址到逻辑地址的抽象、虚拟和映射。

寻址问题解决之后，就应该考虑怎么向磁盘发送需要写入的数据了。针对这个问题，人们抽象出一套接口系统，专门用于计算机和其外设交互数据，称为 SCSI 接口协议，即小型计算机系统接口。

我们举个例子来说明，比如某时刻要向磁盘写入 512 字节的数据，磁盘控制器先向磁盘发一个命令，表明要准备做 IO 操作了，而且说明了附带参数(是否启用磁盘缓存、完成后是否中断通知 CPU 等)，磁盘应答说可以进行，控制器立即将所要 IO 的类型(读/写)和扇区的起始地址以及随后扇区的数量(长度)发送给磁盘，如果是写 IO，则随后还要将需要写入的数据发送给磁盘，磁盘将这块数据顺序写入先前通告的扇区中。

提示

新的 SCSI 标准中有一种促进 IO 效率的新的方式，即 Skip Mask IO 模式。如果有两个 IO，二者 IO 的目标扇区段被隔开了一小段，比如第一个写 IO 的目标为从 1000 开始的随后 128 扇区，第二个写 IO 的目标则为 1500 开始的随后 128 扇区，可以合并这两个 IO 为一个针对 1000 开始的随后 728 个扇区的 IO，控制器将这条指令下发到磁盘之后，还会立即发送一个 Mask 帧，这个帧中包含了一串 bit 流，每一位表示一个扇区，此位为 1，则表示进行该扇区的 IO，为 0，则表示跨过此扇区，不进行 IO。这样，多了这串很小的 bit 流，却能省下一轮额外的 IO 开销。

提示

SCSI 接口完成了访问磁盘过程的虚拟化和抽象，极大的简化了访问磁盘的过程，它屏蔽了磁盘内部结构和逻辑，使得控制器只知道 LBA 是一个房间，有什么数据就给出地址，然后磁盘就会将数据写入这个地址对应的房间，读取操作也一样。

3.3 磁盘相关高层技术

3.3.1 磁盘中的队列技术

想象有一个包含 10000 个同心圆的转盘在旋转，现在有 2 个人在转盘外面，有一个机

械手臂可以将物体放到任何一个同心圆上去。现在，第一个人想到半径最小的同心圆上去，而另外一个人却想到半径最大的同心圆上去，这可让机械手臂犯了难，机械手臂只能按照顺序，先照顾第一个人的要求。它首先寻道到最内侧同心圆，然后转盘旋转到待定位置后将这个人放到轨道上，随后立即驱动磁头臂到最外侧的圆，再将第二个人放上去。这期间的主要时间都用于从内侧到外测的换道过程了，非常浪费。

这只是 2 个人的情况，那么如果有多个人，比如 3 个人先后告诉机械手臂，第一个人说要放到最内侧的圆上，第二个人说要放到最外侧的圆上，第三个人要放到最内侧的圆上。

如果这时候机械手臂还是按照顺序来操作，那么中间就会多了一次无谓的换道操作，极其浪费。所以机械手臂自作主张，在送完第一个人后，它没有立即处理第二个人的请求，而是在脑海中算计，它看第三个人也要求到内侧圆上，而它自己此时也恰好正在内侧圆上，何不趁此捎带第三个人呢？所以磁头跳过第二个人的请求，先把第三个人送到目的地，然后再换道送第二个人。

因为磁头算计用的时间比来回换道快得多，所以这种排队技术大大提高了读写效率。这种例子还有很多，比如电梯就是个很好的例子。实现队列功能的程序控制代码是存放在磁盘控制电路芯片中的，而不是主板上的磁盘控制器上。也就是说，由控制器发给磁盘指令，然后由磁盘自己的 DSP 固化电路或者由磁盘上的微处理器载入代码从而执行指令排队功能。

但是一个巴掌拍不响，排队必须也要由磁盘控制器来支持，所谓的支持就是说，如果磁盘擅自排队，不按照控制器发送过来的顺序一条一条执行指令，则在读出数据之后，由于步调和控制器期望的不一致，预先读出的数据只能先存放到磁盘驱动器的缓存中，等待控制器主动来取。因为控制器给磁盘发送的读写数据的指令，有可能是有先后顺序的，如果磁盘擅自做了排队，将后来发送的指令首先执行，那么读出的数据就算传送给了磁盘控制器，也会造成错乱。

所以，要实现排队技术，仅仅有磁盘驱动器自身是不够的，还必须在磁盘控制器(指主板上的磁盘控制，而不是磁盘本身的控制电路)电路中固化代码处理排队，和磁盘达成一致。或者不使用固化代码方式，而是修改磁盘控制器驱动程序，加入处理排队的功能从而配合磁盘驱动器。



Intel 在 WinHEC 2003 会议上发布了高级主机控制器接口 0.95 版规范(Advanced Host Controller Interface，AHCI)，为驱动程序和系统软件提供了发现并实施命令队列、热插拔及电源管理等高级 SATA 功能的标准接口。这个接口就是在新的控制器硬件之上的驱动层面提供一层接口，解决了磁盘控制器不支持硬盘驱动器自身的排队这个问题。

3.3.2 无序传输技术

还有一种提高磁盘性能的技术，叫做无序数据传输。也就是说，控制器发出一条指令要求读取某些扇区中的内容，磁盘可以不从数据所在的初始扇区开始读，而是采取就近原则。比如，磁头恰好处于待读取数据的尾部，此时如果等待磁盘旋转到磁头位于这块数据

的头部时磁头才开始读，那么时间就被白白地浪费了。如果磁头按照能读多少先读多少的原则，在尾部时就先读出尾部的数据，然后立即发给控制器，控制器立即通过 DMA 将数据放到内存，等磁盘转到数据块头部时再读出剩余的部分发给控制器，这样就避免了时间的浪费。然而，这种技术同样也要由磁盘控制器来支持，或是通过控制器硬件，或是通过驱动程序。

通过指令排队和无序传送可以最大化利用磁盘资源。也就是，把麻烦留给控制器，把简单留给磁盘。因为控制器的处理速度永远比磁盘的机械运动快。

3.3.3 几种可控磁头扫描方式评论

假设目前磁盘控制器的队列中存在如下的一些 IO，这些 IO 所需要查找的磁道号码按照先后排列顺序为：98、183、37、122、14、124、65 和 67，而当前磁头处于 53 号磁道，磁头执行寻道操作有以下几种模式。

1. FCFS(First Come First Serve)

在 FCFS 模式下，磁头完全按照 IO 进入的先后顺序执行寻道操作，即从 53 号磁道跳到 98 号，然后到 183 号，再回到 37 号，依此类推。可以算出在这个例子中，此模式下磁头滑过的磁道总数为 640。对应的扫描图如图 3.8 所示。

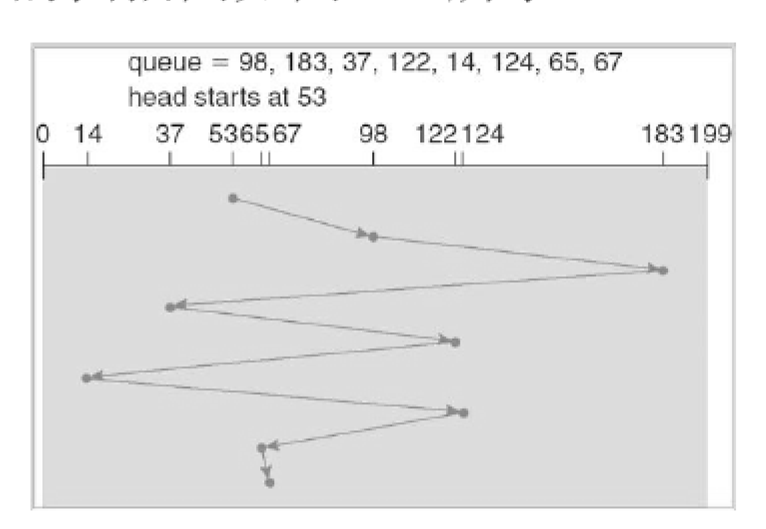


图3.8 FCFS 模式扫描图

显然，FCFS 模式很不科学，在随机 IO 的环境中严重影响 IO 效率。

2. SSTF(Shortest Seek Time First)

在 SSTF 模式下，控制器会优先让磁头跳到离当前磁头位置最近的一个 IO 磁道去读写，读写完毕后，再次跳到离刚读写完的这个磁道最近的一个 IO 磁道去读写，依此类推。在 SSTF 模式下，这个例子呈现的扫描图如图 3.9 所示。

本例中，磁头初始位置在 53 号磁道，如果此时 IO 队列中不断有位于 53 号磁道周围磁道的 IO 进入，比如 55 号磁道、50 号、51 号等，那么诸如 183 号这种离 53 号磁道较远的 IO 将会被饿死，永远也轮不到 183 号磁道的 IO。所以 SSTF 模式的限制也是很大的。

3. SCAN(回旋扫描模式)

这种扫描方式是最传统、最经典的方式了。它类似于电梯模型，从一端到另一端，然后折返，再折返，这样循环下去。磁头从最内侧磁道依次向外圈磁道寻道。然而就像电梯一样，如果这一层没有人等待搭乘，那么磁头就不在本层停止。也就是说如果当前队列中没有某个磁道的 IO 在等待，那么磁头就不会跳到这个磁道上，而是直接略过去。但是 SCAN

模型中，即使最外圈或者最内圈的磁道没有 IO，磁头也要触及到之后才能折返，这就像 50 米往返跑一样，必须触及到终点线才能折返回去。SCAN 模式的扫描图如图 3.10 所示。

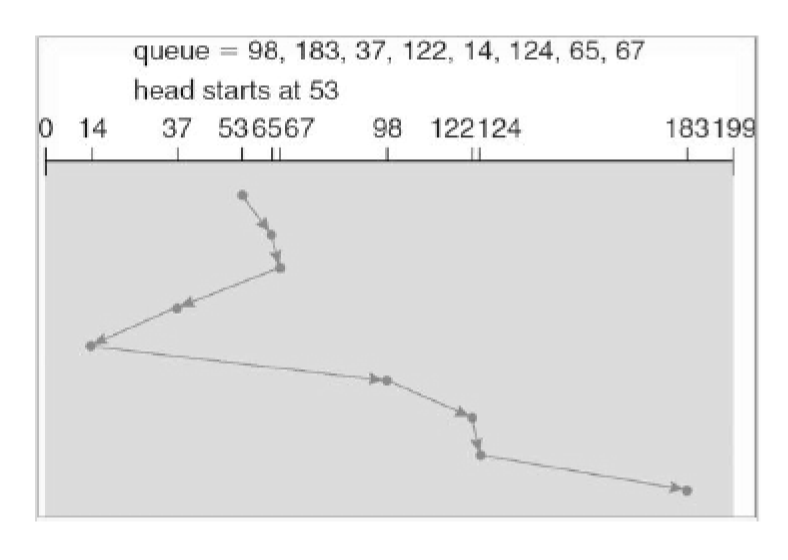


图3.9 SSTF 模式扫描图

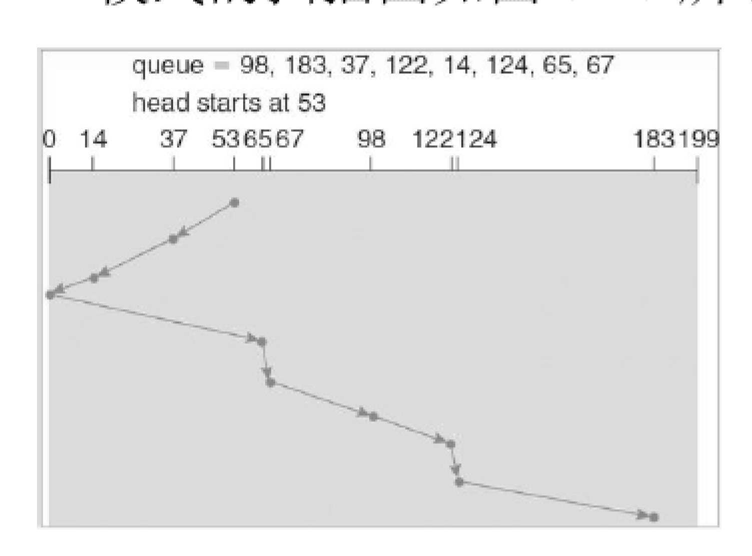


图3.10 SCAN 模式扫描图

SCAN 模式不会饿死任何 IO，每个 IO 都有机会搭乘磁头这个电梯。然而，SCAN 模式也会带来不必要的开销，因为磁头从来不会在中途折返，而只能触及到终点之后才能折返。. 如果磁头正从中间磁道向外圈移动，而此时队列中进入一个内圈磁道的 IO，那么此时磁头并不会折返，即使队列中只有这一个 IO。这个 IO 只能等待磁头触及最外圈之后折返回来被执行。

4. C-SCAN(单向扫描模式)

在 C-SCAN 模式中磁头总是从内圈向外圈扫描，达到外圈之后迅速返回内圈，返回途中不接受任何 IO，然后再从内圈向外圈扫描。C-SCAN 模式的扫描图如图 3.11 所示。

5. LOOK(智能监察扫描模式)和 C-LOOK(智能监察单向扫描模式)

LOOK 模式相对于 SCAN 模式的区别在于，磁头不必达到终点之后才折返，而只要完成最两端的 IO 即可折返。同样，C-LOOK 也是一样的道理，只不过是单向扫描。图 3.12 所示的是 C-LOOK 模式的扫描图。

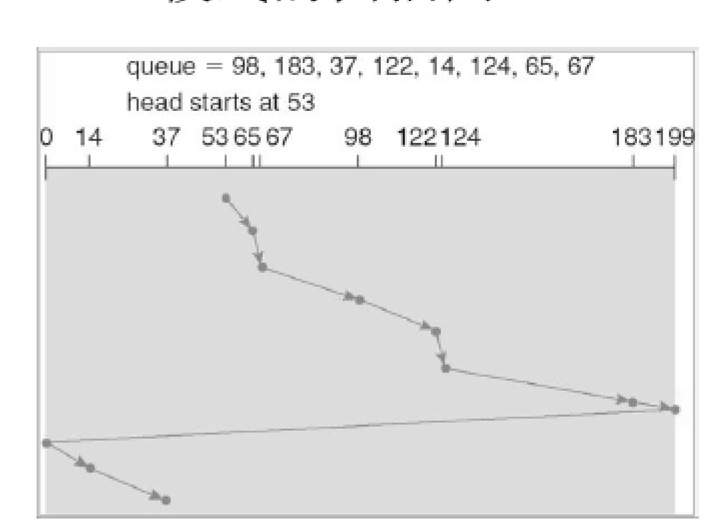


图3.11 C-SCAN 模式扫描图

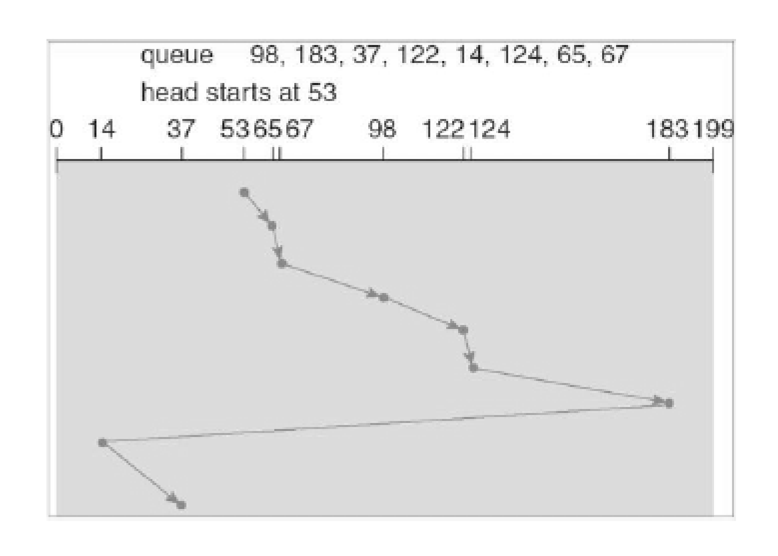


图3.12 C-LOOK 模式扫描图



关于几种扫描模式的选择：总地说来，在负载不高的情况下，SSTF 模式可以获得最佳的性能。但是鉴于可能造成某些较远的 IO 饿死的问题，所以在高负载条件下，SCAN 或者 C-SCAN、C-LOOK 模式更为合适。

在大量随机 IO 的情况下，磁盘的磁头臂会像蜜蜂翅膀一样振动，当然他们的频率可能相差很大，但是用肉眼观察的话，磁头臂确实会像琴弦一样的摆动，频率是比较高的。大家可以去 Internet 上搜索一下磁盘寻道的一个视频，来增强感观认识。

3.3.4 关于磁盘缓存

磁盘上必须有缓存，用来接收指令和数据，还被用来进行预读。磁盘缓存时刻处于打开状态。有很多文档资料上提到某些情况下可以“禁用”磁盘缓存，这是容易造成误解的说法。缓存在磁盘上就表现为一块电路板上的 RAM 芯片，目前有 2MB、8MB、16MB、32MB 等容量规格。所谓“禁用”磁盘缓存指的其实是本书第 5 章中描述的 Write Through 模式，即磁盘收到写入指令和数据后，必须先将其写入盘片，然后才向控制器返回成功信号，这样就相当于“禁用”了缓存。但是实际上，指令和数据首先到达的一定是缓存。

SCSI 指令中有两个参数可以控制对磁盘缓存的使用。

- **DPO(Disable Page Out)**：这个参数的作用是禁止缓存中的数据页(缓存中的数据以页为单位存在)被换出。不管读还是写，被置了这个参数位的数据在缓存空间不够的时候不能覆盖缓存中的其他数据，也就是不能将其他数据换出。
- **FUA(Force Unit Access)**：这个参数的作用是强制盘片访问。对于写操作，磁盘必须将收到的数据写入盘片才返回成功信号，也就是进行 Write Through。对于读操作，磁盘收到指令后，直接去盘片上读取数据，而不搜索缓存。

所以，当某个 SCSI 指令的 DPO 和 FUA 两个参数的值都被设置为 1 时，便相当于完全不使用缓存的提速功能了，但是指令和数据依然会存放到缓存中，这一点需要分清和理解。

3.3.5 影响磁盘性能的因素

目前磁盘可以分为单碟盘和多碟盘，前者在盘体内只有一张盘片，后者则有多张。前面已经讲过，每张盘片的正反两面都可以存放数据，所以每张盘片需要有两个磁头，各读写一面。然而，有一点必须澄清，磁盘每个时刻只允许一个磁头来读写数据。也就是说，不管盘体内盘片和磁头再多，也不可能提高硬盘的吞吐量和 IO 性能，只能提高容量。然而，已经有很多人致力于改变这个现状，希望能让磁头在盘内实现并发读写，也就相当于盘片和盘片之间相互形成 RAID 从而提高性能，但是这项工程目前还没有可以应用的产品。

影响硬盘性能的因素包括如下。

- 转速：转速是影响硬盘连续 IO 时吞吐量性能的首要因素。读写数据时，磁头不会动，全靠盘片的转动来将对应扇区中的数据感应给磁头，所以盘片转得越快，数据传输时间就越短。在连续 IO 情况下，磁头臂寻道次数很少，所以要提高吞吐量或者 IOPS 的值，转速就是首要影响因素了。目前中高端硬盘一般都为 10000 转每秒或者 15000 转每秒。最近也有厂家要实现 20000 转每秒的硬盘，已经有了成形的产品，但是最终是否会被广泛应用，尚待观察。
- 寻道速度：寻道速度是影响磁盘随机 IO 性能的首要因素。随机 IO 情况下，磁头臂需要频繁更换磁道，用于数据传输的时间相对于换道消耗的时间来说是很少的，根本不在一个数量级上。所以如果磁头臂能够以很高的速度更换磁道，那么就会提升随机 IOPS 值。目前高端磁盘平均寻道速度都在 10ms 以下。
- 单碟容量：单碟容量也是影响磁盘性能的一个间接因素。单碟容量越高，证明相同空间内的数据量越大，也就是数据密度越大。在相同的转速和寻道速度条件下，具有高数据密度的硬盘会显示出更高的性能。因为在相同的开销下，单碟容量高

的硬盘会读出更多的数据。目前已有厂家研发出单碟容量超过 300GB 的硬盘，但是还没有投入使用。

- 接口速度：接口速度是影响硬盘性能的一个最不重要的因素。目前的接口速度在理论上都已经满足了磁盘所能达到的最高外部传输带宽。在随机 IO 环境下，接口速度显得更加不重要，因为此时瓶颈几乎全部都在寻道速度上。不过，高端硬盘都用高速接口，这是普遍做法。

3.4 硬盘接口技术

硬盘是个复杂的技术，到目前为至也只有欧洲、美国等发达国家和地区掌握了关键技术能够制造。但不管硬盘内部多么复杂，它必定要给使用者一个简单的接口，用来对其访问读取数据，而不必关心这串数据到底该什么时候写入，写入到哪个盘片，用哪个磁头，等等。

下面就来看一下硬盘向用户提供的是什么样的接口。注意，这里所说的接口不是说物理上的接口，而是包括物理、逻辑在内的抽象出来的接口。也就是说，一个事物面向外部的时候，为达到被人使用的目的而向外提供的一种打开的、抽象的协议，类似于说明书。

目前，硬盘提供的物理接口包括如下。

- 用于 ATA 指令系统的 IDE 接口。
- 用于 ATA 指令系统的 SATA 接口。
- 用于 SCSI 指令系统的并行 SCSI 接口。
- 用于 SCSI 指令系统的串行 SCSI(SAS)接口。
- 用于 SCSI 指令系统的 IBM 专用串行 SCSI 接口(SSA)。
- 用于 SCSI 指令系统的并且承载于 FabreChannel 协议的串行 FC 接口(FCP)。

3.4.1 IDE 硬盘接口

IDE 的英文全称为 Integrated Drive Electronics，即电子集成驱动器，它的本意是指把控制电路和盘片、磁头等放在一个容器中的硬盘驱动器。把盘体与控制电路放在一起的做法减少了硬盘接口的电缆数目与长度，数据传输的可靠性得到了增强。而且硬盘制造起来更加容易，因为硬盘生产厂商不需要再担心自己的硬盘是否与其他厂商生产的控制器兼容。对用户而言，硬盘安装起来也更为方便了。IDE 这一接口技术从诞生至今就一直在不断发展，性能也不断地提高。其拥有价格低廉、兼容性强的特点。IDE 接口技术至今仍然有很多用户，但是正在不断减少。

IDE 接口，也称为 PATA 接口，即 Parallel ATA(并行传输 ATA)。ATA 的英文拼写为 Advanced Technology Attachment，即高级技术附加，貌似发明 ATA 接口的人认为这种接口是有高技术含量的。不过在那个年代应该也算是比较有技术含量的了。ATA 接口最早是在 1986 年由 Compaq、West Digital 等几家公司共同开发的，在 20 世纪 90 年代初开始应用于台式机系统。最初，它使用一个 40 芯电缆与主板上的 ATA 接口进行连接，只能支持两个硬盘，最大容量也被限制在 504MB 之内。后来，随着传输速度和位宽的提高，最后一

代的 ATA 规范使用 80 芯的线缆，其中有一部分是屏蔽线，不传输数据，只是为了屏蔽其他数据线之间的相互干扰。

1. 7 种 ATA 物理接口规范

ATA 接口从诞生至今，共推出了 7 个不同的版本，分别是 ATA-1(IDE)、ATA-2(EIDE Enhanced IDE/Fast ATA)、ATA-3(FastATA-2)、ATA-4(ATA33)、ATA-5(ATA66)、ATA-6(ATA100)和 ATA-7(ATA 133)。

- ATA-1：在主板上有一个插口，支持一个主设备和一个从设备，每个设备的最大容量为 504MB，支持的 PIO-0 模式传输速率只有 3.3MB/s。ATA-1 支持的 PIO 模式包括 PIO-0、PIO-1 和 PIO-2 模式，另外还支持 4 种 DMA 模式(没有得到实际应用)。ATA-1 接口的硬盘大小为 5 英寸，而不是现在主流的 3.5 英寸。
- ATA-2：是对 ATA-1 的扩展，习惯上也称为 EIDE(Enhanced IDE)或 Fast ATA。它在 ATA 的基础上增加了 2 种 PIO 和 2 种 DMA 模式(PIO-3)，不仅将硬盘的最高传输率提高到 16.6MB/s，同时还引进了 LBA 地址转换方式，突破了固有的 504MB 的限制，可以支持最高达 8.1GB 的硬盘。在支持 ATA-2 的 BIOS 设置中，一般可以看到 LBA(Logical Block Address)和 CHS(Cylinder，Head，Sector)设置选项。同时在 EIDE 接口的主板上一般有两个 EIDE 插口，也就是由同一个 ATA 控制器操控的两个 IDE 通道，每个通道可以分别连接一个主设备和一个从设备，这样一块主板就可以支持 4 个 EIDE 设备。这两个 EDIE 接口一般称为 IDE1 和 IDE2。
- ATA-3：没有引入更高速度的传输模式，在传输速度上并没有任何的提升，最高速度仍旧为 16.6MB/s。只在电源管理方案方面进行了修改，引入了简单的密码保护安全方案。同时还引入了一项划时代的技术，那就是 S.M.A.R.T(Self-Monitoring Analysis and Reporting Technology，自监测、分析和报告技术)。这项技术可以对磁头、盘片、电机、电路等硬盘部件进行监测，通过检测电路和主机的监测软件对磁盘进行检测，把其运行状况和历史记录同预设的安全值进行比较分析。当检测到的值超出了安全值的范围时，会自动向用户发出警告，进而对硬盘潜在故障做出有效预测，提高了数据存储的安全性。
- ATA-4：从 ATA-4 接口标准开始正式支持了 Ultra DMA 数据传输模式，因此也习惯称 ATA-4 为 Ultra DMA 33 或 ATA33，33 是指数据传输的速率为 33.3MB/s。并首次在 ATA 接口中采用了 Double Data Rate(双倍数据传输)技术，让接口在一个时钟周期内传输数据两次，时钟上升期和下降期各有一次数据传输，这样数据传输速率一下子从 16.6MB/s 提升至 33.3MB/s。Ultra DMA 33 还引入了冗余校验技术(CRC)。该技术的设计原理是系统与硬盘在进行传输的过程中，随数据一起发送循环的冗余校验码，对方在收取的时候对该校验码进行检验，只有在检验完全正确的情况下才接收并处理得到的数据，这对于高速传输数据的安全性极其有力的保障。
- ATA-5：ATA-5 也就是 Ultra DMA 66，也叫 ATA66，是建立在 Ultra DMA 33 硬盘接口的基础上的，同样采用了 UDMA 技术。Ultra DMA 66 将接口传输电路的频率提高为原来的两倍，所以接收/发送数据速率达到 66.6 MB/s。它保留了 Ultra DMA 33 的核心技术——冗余校验技术。在工作频率提升的同时，电磁干扰问题开始出

现在 ATA 接口中。为保障数据传输的准确性，防止电磁干扰，Ultra DMA 66 接口开始使用 40 针脚 80 芯的电缆。40 针脚是为了兼容以往的 ATA 插槽，减小成本的增加。80 芯中新增的都是信号屏蔽线，这 40 条屏蔽线不与接口相连，所以针脚不需要增加。这种设计可以降低相邻信号线之间的电磁干扰。

- ATA-6：ATA100 接口的数据线与 ATA66 一样，也是使用 40 针 80 芯的数据传输电缆，并且 ATA100 接口完全向下兼容，支持 ATA33 和 ATA66 接口的设备完全可以继续在 ATA100 接口中使用。ATA100 规范将电路的频率又提升了一个等级，可以让硬盘的外部传输率达到 100MB/s。它提高了硬盘数据的完整性与数据传输速率，对桌面系统的磁盘子系统性能有较大的提升作用，而 CRC 技术更有效保证了在高速传输中数据的完整性和可靠性。
- ATA-7：ATA-7 是 ATA 接口的最后一个版本，也叫 ATA133。ATA133 接口支持 133 MB/s 数据传输速度，这是第一种在接口速度上超过 100MB/s 的 IDE 硬盘，迈拓是目前惟一一家推出这种接口标准硬盘的制造商。由于并行传输随着电路频率的提升，传输线缆上的信号干扰越来越难以解决，已经达到了当前技术的极限，所以其他 IDE 硬盘厂商停止了对 IDE 接口的开发，转而生产 Serial ATA 接口标准的硬盘。

图 3.13 几种 ATA 接口的总结。

2. IDE 数据传输模式

- PIO 模式(Programming Input/Output Model)：PIO 模式是一种通过 CPU 执行 I/O 端口指令来进行数据读写的数据交换模式，是最早的硬盘数据传输模式。这种模式的数据传输速率低下，CPU 占有率也很高，传输大量数据时会因为占用过多的 CPU 资源而导致系统停顿，无法进行其他的操作。在 PIO 模式下，硬盘控制器接收到硬盘驱动器传来的数据之后，必须由 CPU 发送信号将这些数据复制到内存中，这就是 PIO 模式高 CPU 占用率的原因。PIO 数据传输模式又分为 PIO mode 0、PIO mode 1、PIO mode 2、PIO mode 3 和 PIO mode 4 几种模式，数据传输速率从 3.3MB/s 到 16.6MB/s 不等。受限于传输速率低下和极高的 CPU 占有率，这种数据传输模式很快就被淘汰了。
- DMA 模式(Direct Memory Access)：直译的意思就是直接内存访问，是一种不经过 CPU 而直接从内存存取数据的数据交换模式。PIO 模式下硬盘和内存之间的数据传输是由 CPU 来控制的，而在 DMA 模式下，CPU 只须向 DMA 控制器下达指令，让 DMA 控制器来处理数据的传送。DMA 控制器直接将数据复制到内存的相应地址上，数据传送完毕后再把信息反馈给 CPU，这样就很大程度上减轻了 CPU 资源的占用率。DMA 模式与 PIO 模式的区别就在于 DMA 模式不过分依赖 CPU，可以大大节省系统资源。二者在传输速度上的差异并不十分明显，DMA 所能达到的最大传输速率也只有 16.6MB/s。DMA 模式可以分为 Single-Word DMA(单字节 DMA)和 Multi-Word DMA(多字节 DMA)两种。
- Ultra DMA 模式(Ultra Direct Memory Access)：一般简写为 UDMA，含义是高级直接内存访问。UDMA 模式采用 16-bit Multi-Word DMA(16 位多字节 DMA)模式为

基准，可以理解为是 DMA 模式的增强版本。它在包含了 DMA 模式的优点的基础上，又增加了 CRC(Cyclic Redundancy Check，循环冗余码校验)技术，提高数据传输过程中的准确性，使数据传输的安全性得到了保障。在以往的硬盘数据传输模式下，一个时钟周期只传输一次数据，而在 UDMA 模式中逐渐应用了 Double Data Rate(双倍数据传输)技术，因此数据传输速度有了极大的提高。此技术就是在时钟的上升期和下降期各自进行一次数据传输，可以使数据传输速度成倍地增长。

可以在 ATA 控制器属性中选择使用 PIO 还是 DMA 传输模式，如图 3.14 所示。

ATA 硬盘接口规格			
接口名称	传输模式	传输速率	电缆
ATA-1	单字节 DMA 0	2.1 MB/s	40 针电缆
	PIO-0	3.3 MB/s	
	单字节 DMA 1，多字节 DMA 0	4.2 MB/s	
	PIO-1	5.2 MB/s	
	PIO-2，单字节 DMA 2	8.3 MB/s	
ATA-2	PIO-3	11.1 MB/s	40 针电缆
	多字节 DMA 1	13.3 MB/s	
	PIO-4，多字节 DMA 2	16.6 MB/s	
ATA-3	PIO-4，多字节 DMA 2	16.6 MB/s	40 针电缆
ATA-4	多字节 DMA3，Ultra DMA 33	33.3 MB/s	40 针电缆
ATA-5	Ultra DMA 66	66.7 MB/s	40 针 80 芯电缆
ATA-6	Ultra DMA 100	100.0 MB/s	40 针 80 芯电缆
ATA-7	Ultra DMA 133	133.0 MB/s	40 针 80 芯电缆

图3.13　几种 ATA 接口总结

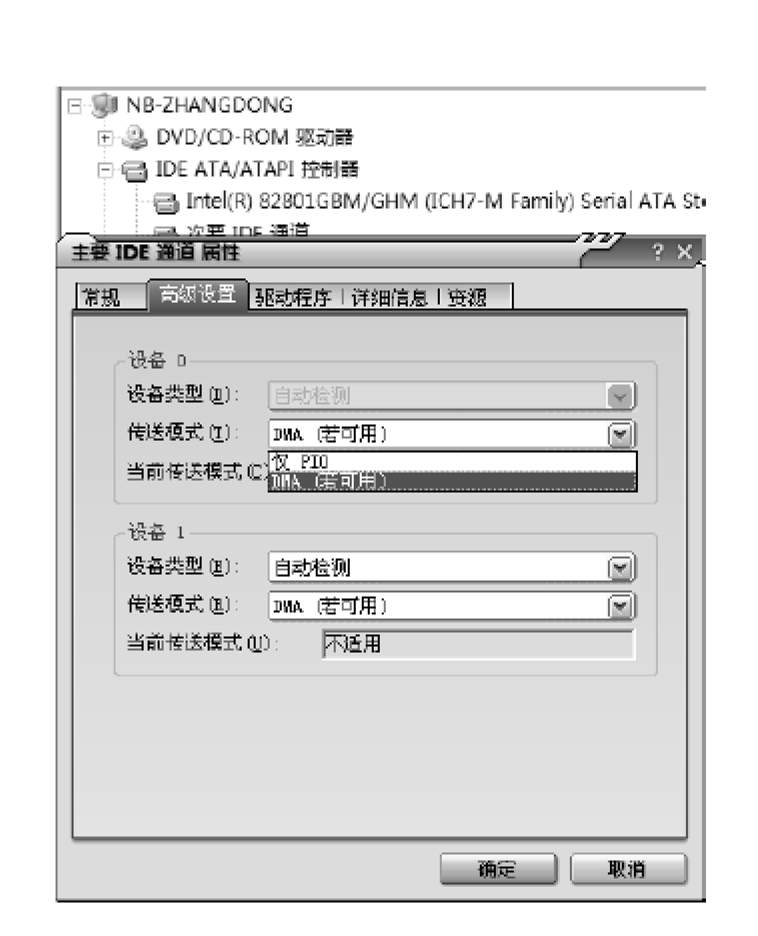


图3.14　DMA 模式示意图

在 UDMA 模式发展到 UDMA133 之后，受限于 IDE 接口的技术规范，无论是连接器、连接电缆、信号协议都表现出了很大的技术瓶颈，而且其支持的最高数据传输率也有限。在 IDE 接口传输率提高的同时，也就是工作频率提高的同时，IDE 接口交叉干扰、地线增多、信号混乱等缺陷也给其发展带来了很大的制约，被新一代的 SATA 接口取代也就在所难免了。

3.4.2　SATA 硬盘接口

SATA 的全称是 Serial ATA，即串行传输 ATA。相对于 PATA 模式的 IDE 接口来说，SATA 是用串行线路传输数据，但是指令集不变，仍然是 ATA 指令集。

SATA 标准是由 Intel、IBM、Dell、APT、Maxtor 和 Seagate 公司共同提出的硬盘接口规范。在 IDF Fall 2001 大会上，Seagate 宣布了 Serial ATA 1.0 标准，正式宣告了 SATA 规范的确立。自 2003 年第二季度 Intel 推出支持 SATA 1.5Gbps 的南桥芯片(ICH5)后，SATA 接口取代传统 PATA 接口的趋势日渐明显。此外，SATA 与现存于 PC 上的 USB、IEEE 1394 相比，在性能和功能方面的表现也更加突出。然而经过一年的市场洗礼，原有的 SATA 1.0/1.0a (1.5Gb/s)规格遇到了一些问题。2005 年 SATA 硬盘步入了新的发展阶段，性能更强、配置更高的 SATA 2.0 产品出现在了市场上，这些高性能的 SATA 2.0 硬盘的到来无疑加速了硬盘市场的转变。

SATA 与 IDE 结构在硬件上有着本质区别，数据接口、电源接口以及接口实物图如图 3.15、图 3.16 及图 3.17 所示。



图3.15　IDE 线缆和 SATA 线缆对比



图3.16　SATA 硬盘的电源线

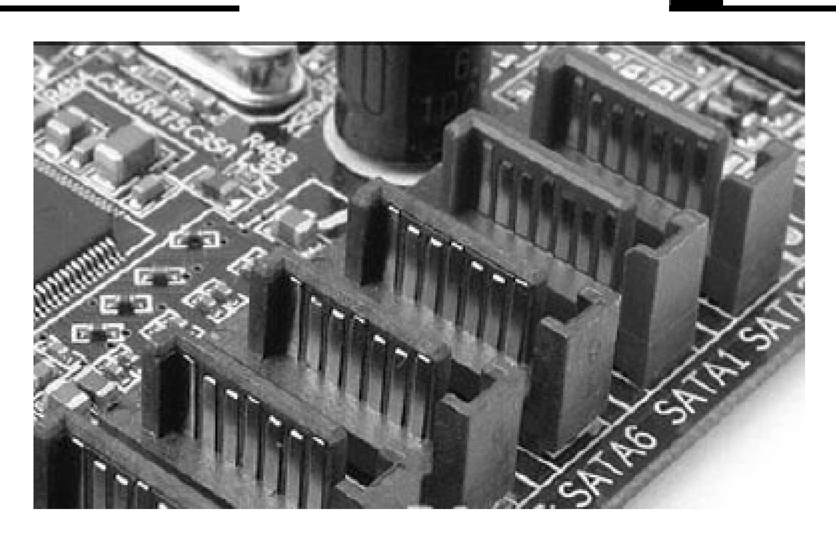


图3.17　SATA 接口实物图

1. SATA 规范发展历程

SATA 技术是 Intel 公司在 IDF 2000 大会上推出的，其最大的优势是传输速率高。SATA 的工作原理非常简单：采用连续串行的方式来实现数据传输从而获得较高的传输速率。2003 年发布的 SATA 1.0 规范提供的传输速率就已经达到了 150MB/s，不但高出普通 IDE 硬盘所提供的 100MB/s(ATA100)，甚至超过了 IDE 最高传输速率 133MB/s(ATA133)。

SATA 在数据可靠性方面也有了大幅度提高。SATA 可同时对指令及数据封包进行循环冗余校验(CRC)，不仅可检测出所有单 bit 和双 bit 的错误，而且根据统计学的原理还能够检测出 99.998%可能出现的错误。相比之下，PATA 只能对来回传输的数据进行校验，而无法对指令进行校验，加上高频率下干扰甚大，因此数据传输稳定性很差。

除了传输速率更高、传输数据更可靠外，节省空间是 SATA 最具吸引力的地方。由于线缆相对于 80 芯的 IDE 线缆来说瘦了不少，更有利于机箱内部的散热，线缆间的串扰也得到了有效控制。不过 SATA 1.0 规范存在不少缺点，特别是缺乏对于服务器和网络存储应用所需的一些先进特性的支持。比如在多任务、多请求的典型服务器环境里面，SATA 1.0 硬盘的确会有性能大幅度下降，还有可维护性不强、可连接性不好等缺点。这时，SATA 2.0 的出现使这方面得到了很好的补充。

2. SATA 2.0 规范中的新特性

与 SATA 1.0 规范相比，SATA 2.0 规范中添加了一些新的特性，具体如下。

- 3Gb/s 的传输速率：在 SATA 2.0 扩展规范中，3Gb/s 的速率是最大的亮点。由于 SATA 使用 8bit/10bit 编码，所以 3Gb/s 等同于 300MB/s 的接口速率。不过，从

性能角度看，3Gb/s 并不能带来多大的提升，即便是 RAID 应用的场合，性能提升也没有想象的那么大。因为硬盘内部传输速率还达不到与接口速率等同的程度。在大多数应用中，硬盘是将更多的时间花在了寻道上，而不是传输上。接口速率的提高直接影响的是从缓存进行读写的操作，所以理论上大缓存的产品会从 3Gb/s 的传输速率中得到更大的好处。

- 支持 NCQ 技术：在 SATA 2.0 扩展规范所带来的一系列新功能中，NCQ(Native Command Queuing，自身命令队列)功能也非常令人关注。硬盘是机电设备，容易受内部机械部件惯性的影响，其中旋转等待时间和寻道等待时间就大大限制了硬盘对数据访问和检索的效率。前面曾经描述过一个模型，指的就是这种由硬盘驱动器自身实现的排队技术。

如果对磁头寻道这个机械动作的执行过程实施智能化的内部管理，就可以大大地提高整个工作流程的效率。所谓智能化的内部管理就是取出队列中的命令，然后重新排序，以便有效地获取和发送主机请求的数据。在硬盘执行某一命令的同时，队列中可以加入新的命令并排在等待执行的作业中。如果新的命令恰好是处理起来机械效率最高的，那么它就是队列中要处理的下一个命令。但有效的排序算法既要考虑目标数据的线性位置，又要考虑其角度位置，并且还要对线性位置和角度位置进行优化，以使总线的服务时间最小，这个过程也称作“基于寻道和旋转优化的命令重新排序”。

台式 PATA 硬盘队列一直被严格地限制为深度不得超过 32 级。如果增加队列深度，可能会起到反作用——增加命令堆积的风险。通常 PATA 硬盘接收命令时有两种选择，一是立即执行命令，二是延迟执行。对于后一种情况，硬盘必须通过设置注意标志和 Service 位来通知主机何时开始执行命令。然而硬盘不能主动与主机通信，这就需要主机定期轮回查询，发现 Service 位后将发出一条 Service 命令，然后才能从硬盘处获得将执行哪一条待执行命令的信息。而且 Service 位不包含任何对即将执行命令的识别信息，所必需的命令识别信息是以标记值的形式与数据请求一同传输的，并仅供主机用于设置 DMA 引擎和接收数据缓冲区。这样主机就不能预先掌握硬盘所设置的辅助位是哪条命令设置的，数据传输周期开始前也无法设置 DMA 引擎，这最终导致了 PATA 硬盘效率低下。

NCQ 包含如下两部分内容。

- 一方面，硬盘本身必须有能力针对实体数据的扇区分布对命令缓冲区中的读写命令进行排序。同时硬盘内部队列中的命令可以随着必要的跟踪机制动态地重新调整或排序，其中跟踪机制用于掌握待执行和已完成作业的情况，而命令排队功能还可以使主机在设备对命令进行排队的时候，断开与硬盘间的连接以释放总线。一旦硬盘准备就绪，就重新连接到主机，尽可能以最快的速率传输数据，从而消除占用总线的现象。
- 另一方面，通信协议的支持也相当重要。因为以前的 PATA 硬盘在传输数据时很容易造成中断，这会降低主控器的效率，所以 NCQ 规范中定义了中断聚集机制。相当于一次执行完数个命令后，再对主控器回传执行完毕的信息，改善处理队列命令的效能。

从最早的希捷 7200.7 系列硬盘开始，NCQ 技术应用于桌面产品的时间至今已超过半年，不过目前 NCQ 对个人桌面应用并没有带来多大的性能提升，在某些情况下还会引起副作用。

而且不同硬盘厂商的 NCQ 方案存在着差异，带来的效果也不同。

- 端口选择器(Port Selector)：目前的 SATA 2.0 扩展规范还具备了 Port Selector(端口选择器)功能。Port Selector 是一种数据冗余保护方案，使用 Port Selector 可增加冗余度具有 Port Selector 功能的 SATA 硬盘，外部有两个 SATA 接口，同时连接这两个接口到控制器上，一旦某个接口坏掉或者连线故障，则立刻切换到另一个接口和连线上，不会影响数据传输。
- 端口复用器(Port Multiplier)：SATA 1.0 的一个缺点就是可连接性不好，即连接多个硬盘的扩展性不好。因为在 SATA 1.0 规范中，一个 SATA 接口只能连接一个设备。SATA 规范的制定者们显然也意识到了这个问题，于是在 SATA 2.0 中引入了 Port Multiplier 的概念。Port Multiplier 是一种可以在一个控制器上扩展多个 SATA 设备的技术，它采用 4 位(bit)宽度的 Port Multiplier 端口字段，其中控制端口占用一个地址，因此最多能输出 2 的四次方减 1 个，即 15 个设备连接，这与并行 SCSI 相当。Port Multiplier 的上行端口只有 1 个，在带宽为 150MB/s 的时候容易成为瓶颈，但如果上行端口支持 300MB/s 的带宽，就与 Ultra320 SCSI 的 320MB/s 十分接近了。Port Multiplier 技术对需要多硬盘的用户很有用，不过目前提供这种功能的芯片组极少。
- 服务器特性：在 SATA 2.0 扩展规范中还增加了大量的新功能，比如防止开机时多硬盘同时启动带来太大电流负荷的交错启动功能；强大的温度控制、风扇控制和环境管理；背板互联和热插拔功能等。这些功能更侧重于低端服务器方面的扩展。
- 接口和连线的强化：作为一个还在不断添加内容的标准集合，SATA 2.0 最新的热点是 eSATA，即外置设备的 SATA 接口标准，采用了屏蔽性能更好的两米长连接线，目标是最终取代 USB 和 IEEE 1394。在内部接口方面，Click Connect 加强了连接的可靠性，在接上时有提示声，拔下时需要先按下卡口。这些细微的结构变化显示出 SATA 接口更加成熟和可靠。

下面以单独的一节来讲解在存储方面应用最为广泛的 SCSI 硬盘接口。

3.5 SCSI 硬盘接口

SCSI 与 ATA 是目前现行的两大主机与外设通信的协议规范，而且它们各自都有自己的物理接口定义。对于 ATA 协议，对应的就是 IDE 接口；对于 SCSI 协议，对应的就是 SCSI 接口。凡是作为一个通信协议，就可以按照 OSI 模型(本书第 7 章将介绍)来将其划分层次，尽管有些层次可能是合并的或者是缺失的。划分了层次之后，我们就可以把这个协议进行分解，提取每个层次的功能和各个层次之间的接口，从而可以将这个协议融合到其他协议之中，形成一种“杂交”协议来适应各种不同的环境，这个话题将在本书第 13 章加以阐述。

SCSI 的全称是 Small Computer System Interface，即小型计算机系统接口，是一种较为特殊的接口总线，具备与多种类型的外设进行通信的能力，比如硬盘、CD-ROM、磁带机和扫描仪等。SCSI 采用 ASPI(高级 SCSI 编程接口)的标准软件接口使驱动器和计算机内部安装的 SCSI 适配器进行通信。SCSI 接口是一种广泛应用于小型机上的高速数据传输技术。SCSI 接口具有应用范围广、多任务、带宽大、CPU 占用率低以及热插拔等优点。

SCSI 接口为存储产品提供了强大、灵活的连接方式，还提供了很高的性能，可以有 8 个或更多(最多 16 个)的 SCSI 设备连接在一个 SCSI 通道上，其缺点是价格过于昂贵。SCSI 接口的设备一般需要配合价格不菲的 SCSI 卡一起使用(如果主板上已经集成了 SCSI 控制器，则不需要额外的适配器)，而且 SCSI 接口的设备在安装、设置时比较麻烦，所以远远不如 IDE 设备使用广泛。虽然从 2007 年开始，IDE 硬盘就被 SATA 硬盘彻底逐出了市场。

在系统中应用 SCSI 必须要有专门的 SCSI 控制器，也就是一块 SCSI 控制卡，才能支持 SCSI 设备，这与 IDE 硬盘不同。在 SCSI 控制器上有一个相当于 CPU 的芯片，它对 SCSI 设备进行控制，能处理大部分的工作，减少了 CPU 的负担(CPU 占用率)。在同时期的硬盘中，SCSI 硬盘的转速、缓存容量、数据传输速率都要高于 IDE 硬盘，因此更多是应用于商业领域。

下面简单介绍一下 SCSI 规范的发展过程。

SCSI 最早是 1979 年由美国的 Shugart 公司(希捷公司前身)制订的，在 1986 年获得了 ANSI(美国标准协会)的承认，称为 SASI(Shugart Associates System Interface)，也就是最初版本 SCSI-1。

SCSI-1 是第一个 SCSI 标准，支持同步和异步 SCSI 外围设备；使用 8 位的通道宽度；最多允许连接 7 个设备；异步传输时的频率为 3MB/s，同步传输时的频率为 5MB/s；支持 WORM 外围设备。它采用 25 针接口，因此在连接到 SCSI 卡(SCSI 卡上接口为 50 针)上时，必须要有一个内部的 25 针对 50 针的接口电缆。该种接口已基本被淘汰，在相当古老的设备上或个别扫描仪设备上可能还可以看到。

SCSI-2 又被称为 Fast SCSI，它在 SCSI-1 的基础上做出了很大的改进，增加了可靠性，数据传输率也被提高到了 10MB/s，但仍旧使用 8 位的并行数据传输，还是最多连接 7 个设备。后来又进行了改进，推出了支持 16 位并行数据传输的 WIDE-SCSI-2(宽带)和 FAST-WIDE-SCSI-2(快速宽带)。其中 WIDE-SCSI-2 的数据传输速率并没有提高，只是改用 16 位传输；而 FAST-WIDE-SCSI-2 则是把数据传输速率提高到了 20MB/s。

SCSI-3 标准版本是在 1995 年推出的，也习惯称为 Ultra SCSI，其同步数据传输速率为 20MB/s。若使用 16 位传输的 Wide 模式时，数据传输率更可以提高至 40MB/s。其允许接口电缆的最大长度为 1.5 米。

1997 年推出了 Ultra 2 SCSI(Fast－40)标准版本，其数据通道宽度仍为 8 位，但其采用了 LVD(Low Voltage Differential，低电平微分)传输模式，传输速率为 40MB/s，允许接口电缆的最大长度为 12 米，大大增加了设备的灵活性，且支持同时挂接 7 个设备。随后推出了 Wide Ultra 2 SCSI 接口标准，它采用 16 位数据通道带宽，最高传输速率可达 80MB/s，允许接口电缆的最大长度为 12 米，支持同时挂接 15 个装置。

LVD 可以使用更低的电压，因此可以将差动驱动程序和接收程序集成到硬盘的板载 SCSI 控制器中。不再需要单独的高成本外部高电压差动组件。而老式 SCSI 需要使用独立的、耗电的高压器件。

LVD 硬盘可进行多模式转换。当所有条件都满足时，硬盘就工作在 LVD 模式下；反之，如果并非所有条件都满足，硬盘将降为单端工作模式。LVD 硬盘带宽的增加对于服务器环境来说意味着更理想的性能。服务器环境都有快速响应、必须能够进行随机访问和大工作量的队列操作等要求。当使用诸如 CAD、CAM、数字视频和各种 RAID 等软件的时候，

带宽增加的效果能立竿见影，信息可以迅速而轻松地进行传输。

Ultra 160 SCSI，也称为 Ultra 3 SCSI LVD，是一种比较成熟的 SCSI 接口标准，是在 Ultra 2 SCSI 的基础上发展起来的，采用了双转换时钟控制、循环冗余码校验和域名确认等新技术。在增强了可靠性和易管理性的同时，Ultra 160 SCSI 的传输速率为 Ultra 2 SCSI 的 2 倍，达到 160MB/s。这是采用了双转换时针控制的结果。双转换时钟控制在不提高接口时钟频率的情况下使数据传输率提高了一倍，这也是 Ultral 60 SCSI 接口速率大幅提高的关键。

Ultra 320 SCSI，也称为 Ultra 4 SCSI LVD，是比较新型的 SCSI 接口标准。Ultra 320 SCSI 是在 Ultra 160 SCSI 的基础上发展起来的，Ultra 160 SCSI 的 3 项关键技术，即双转换时钟控制、循环冗余码校验和域名确认，都得到了保留。以往的 SCSI 接口标准中，SCSI 接口支持异步和同步两种传输模式。Ultra 320 SCSI 引入了调步传输模式，在这种传输模式中简化了数据时钟逻辑，使 Ultra 320 SCSI 的高传输速率成为可能。Ultra 320 SCSI 传输速率可以达到 320MB/s。

图 3.11 为 SCSI 总线连接示意图。

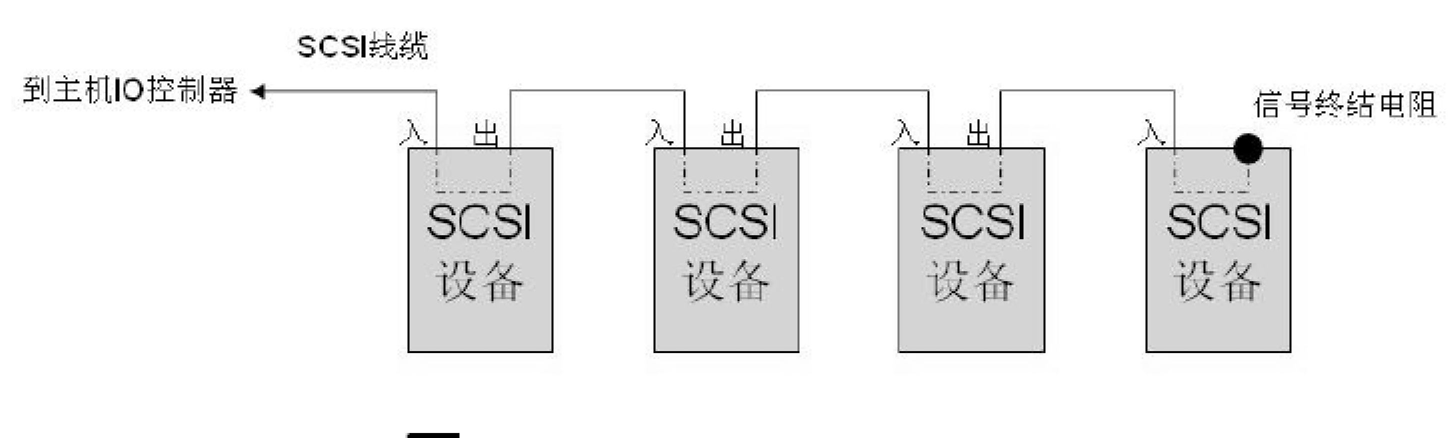


图3.18 SCSI 总线连接示意图

SCSI 协议中的 OSI 模型

上面描述的 SCSI 接口的各个规范，全部限于物理电气层，即描述传输速率、电气技术性能等。而我们也说过，SCSI 是一套完整的数据传输协议。一个通信协议必然会跨越 OSI 的所有 7 个层次，而物理电气参数只是 OSI 模型中的第一层，那么第二层到第七层，SCSI 规范中也包含么？答案当然是肯定的。

1. SCSI 协议的链路层

OSI 模型中链路层的功能就是用来将数据帧成功地传送到这条线路的对端。SCSI 协议中，利用 CRC 校验码来校验每个指令或者数据的帧，如果发现对方发来的校验码与本地计算的不同，则说明这个数据帧在传输过程中受到了比较强的干扰而使其中某个或者某些位发生了翻转，那么就会丢弃这个帧，发送方便会重传这个帧。

2. SCSI 协议的网络层

1) SCSI 总线编址机制

OSI 模型中网络层的功能就是用来寻址的，那么面对总线或者交换架构下的多个节点，各个节点之间又是如何区分对方呢？只有解决了这个问题，才能继续，否则是没有意义的。SCSI 协议利用了一个 SCSI ID 的概念来区分每个节点。在 Ultra 320 SCSI 协议中，一条 SCSI 总线上可以存在 16 个节点，其中 SCSI 控制器占用一个节点，SCSI ID 被恒定设置为 7。其他 15 个节点的 SCSI ID 可以随便设置但是不能重复。这 16 个 ID 中，7 具有最高的优先级。

也就是说，如果 7 这个 ID 要发起传输，则其他 15 个 ID 都必须乖乖把总线的使用权让给它。图 3.19 是 SCSI 总线 ID 优先级示意图。

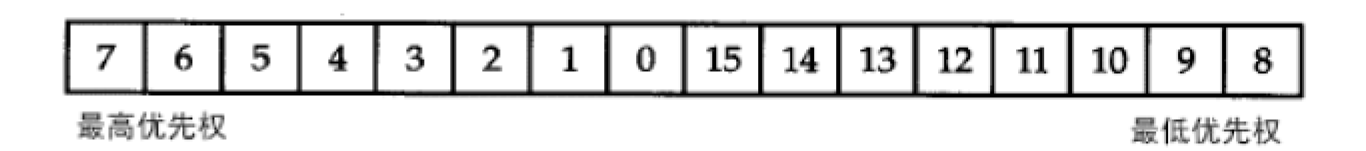


图3.19 SCSI 总线 ID 优先级示意图

由于总线是一种共享的线路，总线上的每个节点都会同时感知到这条线路上的电位信号，所以同一时刻只能由一个节点向这条总线上放数据，也就是给这条线路加一个高电位或者低电位。其他所有节点都能感知到这个电位的增降，但是只有接受方节点才会将感知到的电位增降信号保存到自己的缓存中，这些保存下来的信号就是数据。电路上是高电位则接受方会保存为 1，低电位则保存成 0，反过来保存也可以。图 3.20 是一个只有两条导线的总线网络(实际中的总线远远不止两条导线)，其中总线终结电阻的作用是终结导线上的电位信号。

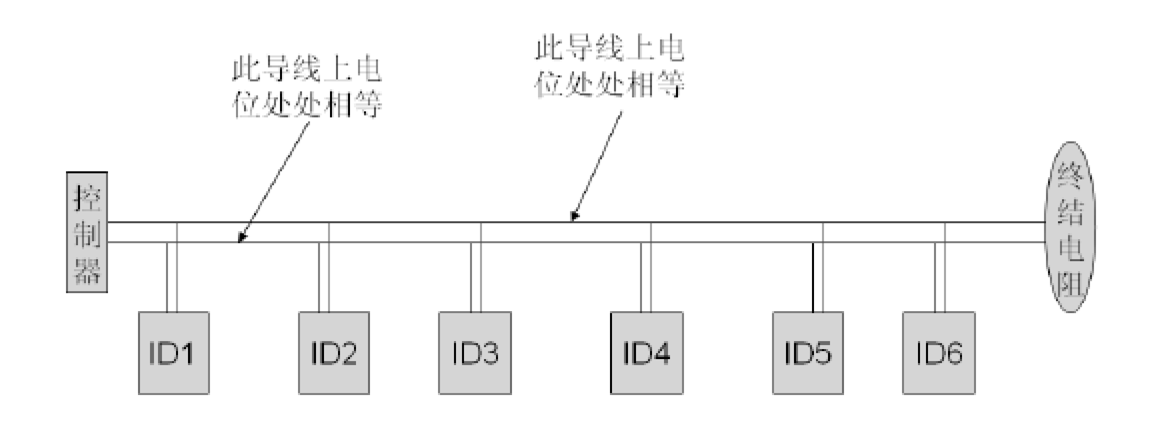


图3.20 SCSI 总线 ID

图 3.21 是一个 32 位数据总线的 SCSI ID 与其优先级以及导线的对应表。

SCSI address	DB31							DB24	DB23							DB16	DB15							DB8	DB7							DB0	Priority
7	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	1
6	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	2
5	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	3
4	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	4
3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	5
2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	6
1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	7
0	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	8
15	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	9
14	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	10
13	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	11
12	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	12
11	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	13
10	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	14
9	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	15
8	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	16
23	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	17
22	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	18
21	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	19
20	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	20
19	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	21
18	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	22
17	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	23
16	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	24
31	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	25
30	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	26
29	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	27
28	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	28
27	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	29
26	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	30
25	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	31
24	-	-	-	-	-	-	-	1	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	32

图3.21 SCSI ID 与其优先级以及导线的对应表

那么这些节点是如何知道现在正在通信的两个节点之中有没有自己呢？要了解当前线路上是不是自己在通信，或者自己想争夺线路的使用权而通告其他节点，这个过程叫做仲

裁。有总线的地方就有仲裁，因为总线是共享的，各个节点都申请使用，所以必须有一个仲裁机制。SCSI 接口并不只有 8 或者 16 条数据线，还有很多控制信号线。

普通台式机主板一般不集成 SCSI 控制器，如果想接入 SCSI 磁盘，则必须增加 SCSI 卡。SCSI 卡一端接入主机的 PCI 总线，另一端用一个 SCSI 控制器接入 SCSI 总线。卡上有自己的 CPU(频率很低，一般为 RISC 架构)，通过执行 ROM 中的代码来控制整个 SCSI 卡的工作。经过这样的架构，SCSI 卡将 SCSI 总线上的所有设备，经过 PCI 总线传递给内存中运行着的 SCSI 卡的驱动程序，这样操作系统便会知道 SCSI 总线上的所有设备了。如果这块卡有不止一个 SCSI 控制器，则每个控制器都可以单独掌管一条 SCSI 总线，这就是多通道 SCSI 卡。通道越多，一张卡可接入的 SCSI 设备就越多。

图 3.22 是 SCSI 总线接入计算机总线的示意图。

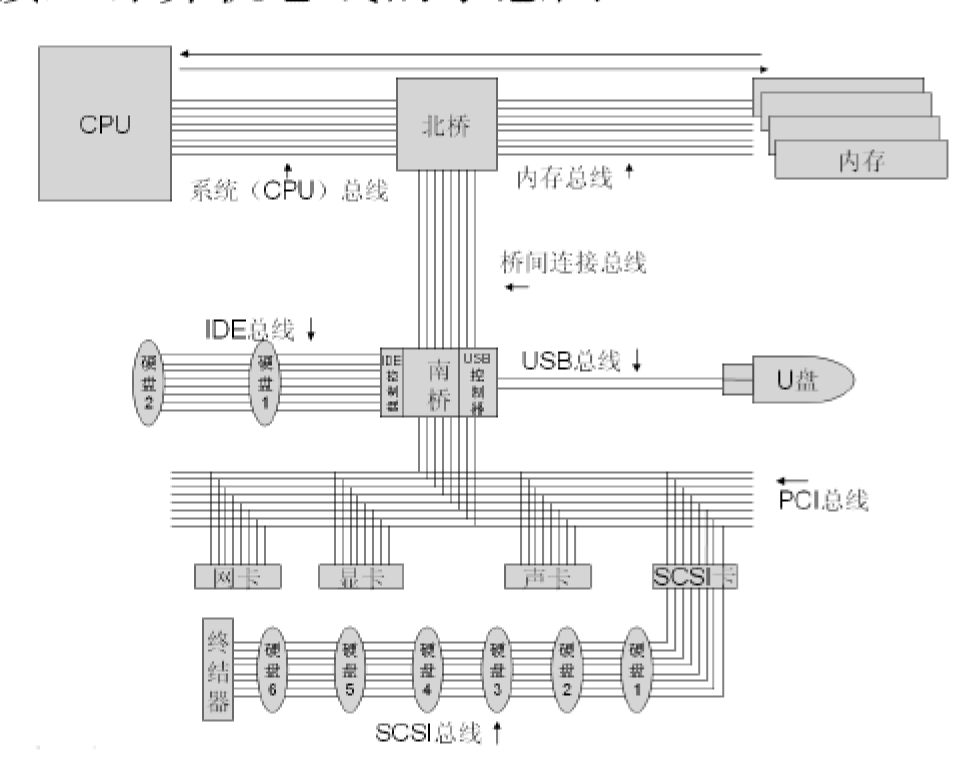


图3.22 SCSI 总线与计算机总线

2) SCSI 寻址机制和几个阶段

- 空闲阶段

  总线一开始是处于一种空闲状态，没有节点要发起通信。总线空闲的时候，BSY 和 SEL 这两条控制信号线的状态都为 False 状态(用一个持续的电位表示)，此时任何节点都可以发起通信。

- 仲裁阶段

  节点申请总线的使用权是通过在 8 条或者 16 条数据总线上(8 位宽 SCSI 有 8 条，16 位宽 SCSI 有 16 条)提升自己对应的那条线路的电位来申请总线使用权的。提升自己 ID 对应线路的电位的同时，这个节点也提升 BSY 线路的电位。每个 ID 号对应这 8 条或者 16 条线中的一条。SCSI 设备上都有跳线用来设置这个设备的 ID 号。跳线设置好之后，这个设备每次申请仲裁都只会在 SCSI 接口的 8 条或 16 条数据线中的对应它自身 ID 的那条线上提升电位。如果同时有多个节点提升了各自线路上的电位，那么所有发起申请的节点均判断总线上的这些信号，如果自己是最高优先级的，那么就持续保留这个信号。而其他低优先级的节点一旦检测到高优先级的 ID 线路上有信号，则立即撤销自身的信号，回到初始状态等待下轮仲裁，而最高优先级的 ID 就在这轮仲裁中获胜，取得总线的使用权，同时将 SEL 信号线提升电位。

  SCSI 总线的寻址方式，按照控制器－通道－SCSI ID－LUN ID 来寻址。LUN 是个新名词，全称是 Logical Unit Number，下面内容中我们会对它进行描述。

先看一下控制器一级寻址。控制器就是指 SCSI 控制器，这个控制器集成在南桥上，或者独立于某个 PCI 插卡。但不管在哪里，它们都要连接到主机 IO 总线上。有 IO 端口，就可以让 CPU 访问到。一个主机 IO 总线上不一定只有一个 SCSI 控制器，可以有多个，比如插入多张 SCSI 卡到主板，那么就会在 Windows 系统的设备管理器中发现多个 SCSI 控制器。系统会区分每个控制器。

每个控制器又可以有多个通道。通道也就是 SCSI 总线，一条 SCSI 总线就是一个通道。那么多条 SCSI 总线(通道)可以被一个控制器管理么？答案是肯定的，这个物理控制器会被逻辑划分为多个虚拟的，可以管理多个通道(SCSI 总线)的控制器，称为多通道控制器。目前市场上有的产品可以将 4 个通道集成到一个单独的 SCSI 卡上。不仅仅 SCSI 控制器可以有多通道，IDE 控制器也有通道的概念。我们知道，普通台式机主板上一般会有两个 IDE 插槽，一个 IDE 插槽可以连接两个 IDE 设备，但是设备管理器中只有一个 IDE 控制器，如图 3.23 所示。也就是说一个控制器掌管着两个通道，每个通道(总线)上都可以接入两个 IDE 设备。



图3.23 Windows 中的 IDE 控制器

常说的“单通道 SCSI 卡”和“双通道 SCSI 卡”，就是指上面可以接几条 SCSI 总线。当然通道数目越多，能接入的 SCSI 设备也就越多。

每个通道(总线)上可以接入 8 或 16 个 SCSI 设备，所以必须区分开每个 SCSI 设备。SCSI ID 就是针对每个设备的编号，每个通道上的设备都有自己的 ID。不同通道之间的设备 ID 可以相同，并不影响它们的区分，因为它们的通道号不同。如图 3.24 为多通道控制器示意图。

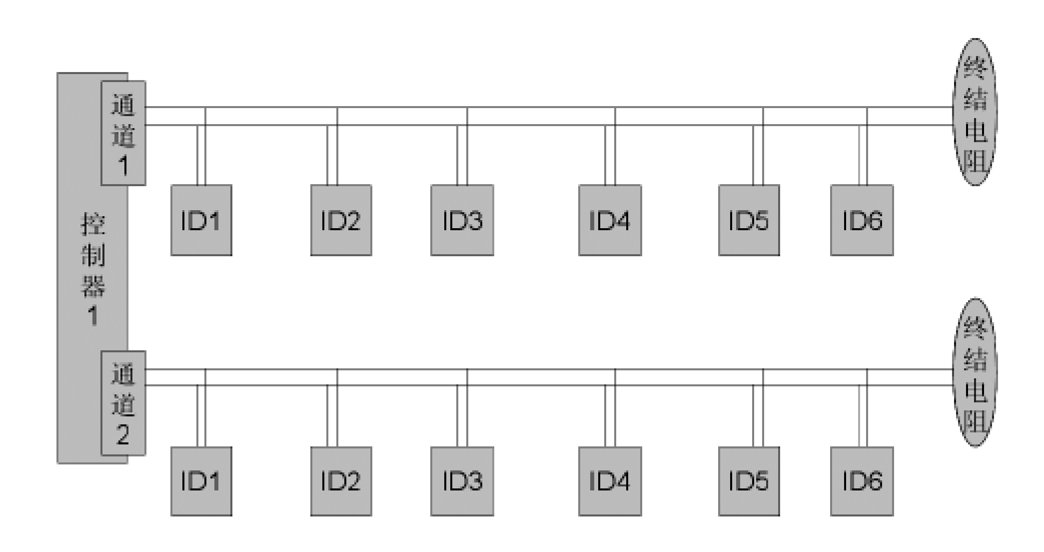


图3.24 多通道控制器示意图

图 3.25 中所示的机器安装了一块 LSI 的 SCSI 卡，但是显示为两个设备，这两个设备就是两个通道。



图3.25 Windows 中的 SCSI 控制器

其中一个在第 3 号 PCI 总线上的第三个设备(第三个 PCI 插槽)上。功能 0 指的就是 0 号通道，如图 3.26 所示。

另一个也在第 3 号 PCI 总线上的第三个设备(第三个 PCI 插槽)上，表明这个设备也

在同一块 SCSI 卡上。功能 1 指的是 1 号通道，如图 3.27 所示。

LSI Adapter, Ultra320 SCSI 2000 series, (with 1020/1030)
设备类型: SCSI 和 RAID 控制器
制造商: LSI Logic
位置: PCI 总线 3、设备 3、功能 0

图3.26 Windows 中的控制器通道号(1)

LSI Adapter, Ultra320 SCSI 2000 series, (with 1020/1030)
设备类型: SCSI 和 RAID 控制器
制造商: LSI Logic
位置: PCI 总线 3、设备 3、功能 1

图3.27 Windows 中的控制器通道号(2)

然而，SCSI ID 并不是 SCSI 总线网络中的最后一层地址，还有一个 LUN ID。这个是做什么用的呢？难道一个 SCSI ID，也就是一个 SCSI 设备，还可以再划分？是的，可以再分！再分就不是物理上的分割了，总不能把一个 SCSI 设备掰开两半吧。只能在逻辑上分，每个 SCSI ID 下面可以再区分出来若干个 LUN ID。控制器初始化的时候，会对每个 SCSI ID 上的设备发出一条 Report Lun 指令，用来收集每个 SCSI ID 设备的 LUN 信息。这样，一条 SCSI 总线上可接入的最终逻辑存储单元数量就大大增加了。LUN 对传统的 SCSI 总线来说意义不大，因为传统 SCSI 设备本身已经不可物理上再分了。如果一个物理设备上没有再次划分的逻辑单元，那么这个物理设备必须向控制器报告一个 LUN0，代表物理设备本身。对于带 RAID 功能的 SCSI 接口磁盘阵列设备来说，由于会产生很多的虚拟磁盘，所以只靠 SCSI ID 是不够的，这时候就要用到 LUN 来扩充可寻址的范围，所以习惯上称磁盘阵列生成的虚拟磁盘为 LUN。关于 RAID 和磁盘阵列会分别在本书的第 4 章到第 6 章中介绍。

- 选择阶段

  仲裁阶段之后，获胜的节点会将 BSY 和 SEL 信号线置位，然后将 8 或 16 条数据总线上对应它自身 ID 的线路和对应它要通信的目标 ID 的线路的电位提升，这样目的节点就能感知到它自己的线路上来了信号，开始做接收准备。



SCSI 控制器也是总线上的一个节点，它的优先级必须是最高的，即等于 7，因为控制器需要掌控整条总线。

总线上最常发生的是控制器向其他节点发送和接收数据，而除控制器之外的其他节点之间交互数据，一般是不会发生的。如果要从总线上的一块硬盘复制数据到另一块硬盘，那也必须先将数据发送到控制器，控制器再复制到内存，经过 CPU 运算后再次发给控制器，然后控制器再发给另外一块硬盘。经过这么长的路径而不直接让这两块硬盘建立通信，原因就是硬盘本身是不能感知文件这个概念的，硬盘只理解 SCSI 语言，而 SCSI 语言是处理硬盘 LBA 块的，即告诉硬盘读或者写某些 LBA 地址上的扇区(块)，而不可能告诉硬盘读写某个文件。文件这个层次的功能是由运行在主机上的文件系统代码所实现的，所以硬盘必须将数据先传送到主机内存由文件系统处理，然后再发向另外的硬盘。

这就是 SCSI 的网络层。每个节点都在有条不紊地和控制器交互着数据。

3. SCSI 协议的传输层

OSI 模型中的传输层的功能就是保障此端的数据成功地传送到彼端。与链路层不同的是，链路层只是保障线路两端数据的传送，而且一旦某个帧出错，链路层程序本身不会重新传送这个帧。所以，需要有一个端到端的机制来保障传输，这个机制是运行在通信双方

最终的两端的，而不是某个链路的两端。

图 3.28 显示了 SCSI 协议是如何保障每个指令都被成功传送到对方的。

1】 发起方在获得总线仲裁之后，会发送一个 SCSI Command 写命令帧，其中包含对应的 LUN 号以及 LBA 地址段。接收端接收后，就知道下一步对方就要传输数据了。接收方做好准备后，向发起方发送一个 XFER_RDY 帧，表示已经做好接收准备，可以随时发送数据。

2】 发起方收到 XFER_RDY 帧之后，会立即发送数据。每发送一帧数据，接收方就回送一个 XFER_RDY 帧，表示上一帧成功收到并且无错误，可以立即发送下一帧，直到数据发送结束。

3】 接收方发送一个 RESPONSE 帧来表示这条 SCSI 命令执行完毕。

图 3.29 是一个 SCSI 读过程的示意图。

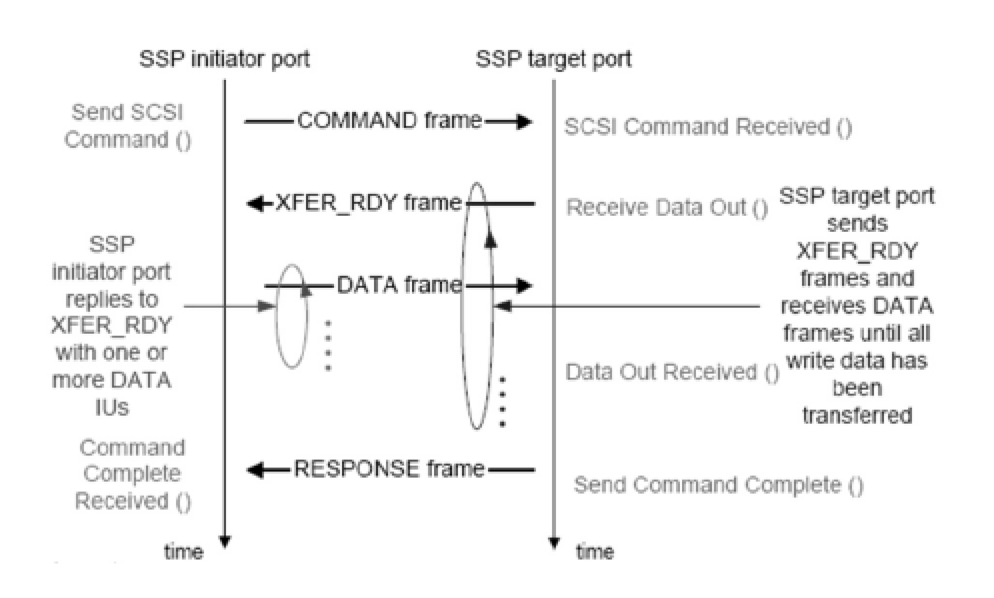


图3.28 控制器向设备发送数据(写入数据)

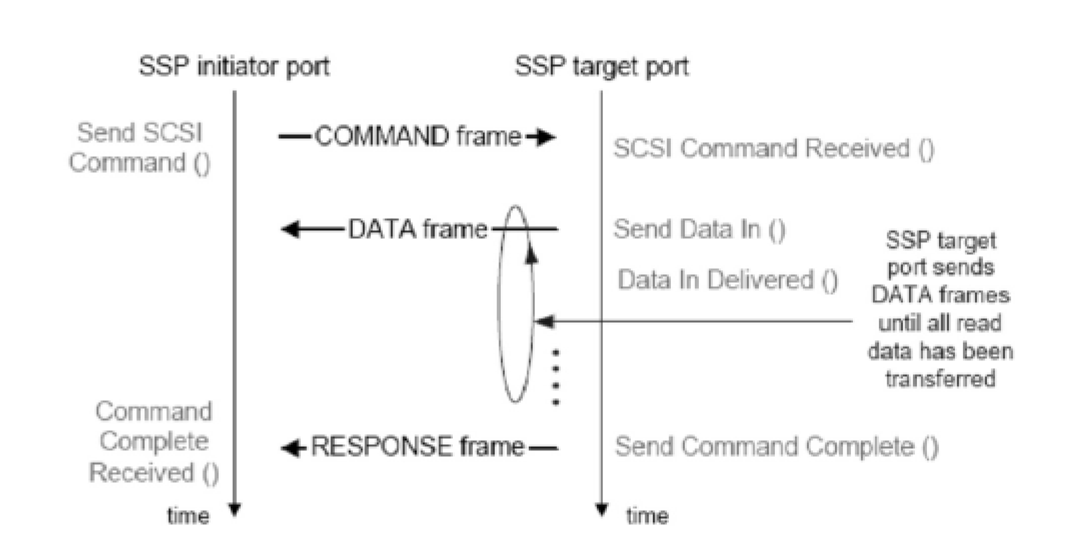


图3.29 控制器向设备读取数据

1】 发起方在获得总线仲裁之后，会发送一个 SCSI Command 读命令帧。接收端接收后，立即将该命令中给出的 LUN 以及 LBA 地址段的所有扇区的数据读出，传送给发起端。

2】 所有数据传输结束后，目标端发送一个 RESPONSE 帧来表示这条 SCSI 命令执行完毕。

SCSI 协议语言就是利用这种两端节点之间相互传送一些控制帧，来达到保障数据成功传输的目的。

4. SCSI 协议的会话层、表示层和应用层

会话层、表示层和应用层是 OSI 模型中最上面的三层，是与底层网络通信语言无关的，底层语言没有必要了解上层语言的含义。有没有会话层，完全取决于利用这个协议进行通信的应用程序。这里我们就不再详述了。

3.6 磁盘控制器、驱动器控制电路和磁盘控制器驱动程序

3.6.1 磁盘控制器

硬盘的接口包括了物理接口，也就是硬盘接入到磁盘控制器上需要用的接口，具体的针数、某个针的作用等。除了物理接口规范之外，还定义了一套指令系统，叫做逻辑接口。磁盘通过物理线缆和接口连接到磁盘控制器之后，若想在磁盘上存放一个字母应该怎么操

作？这是需要业界定义的很重要的东西。指令集定义了“怎样向磁盘发送数据和从磁盘读取数据”。但是这套指令集，不是由 CPU 直接执行代码来生成指令的，而是由专门的芯片或者集成到南桥上的某个部分来负责的，这就是磁盘控制器，如 ATA 控制器或 SCSI 控制器。磁盘控制器的作用是参与底层的总线初始化、仲裁等过程，将这些太过底层的机制过滤掉，从而向驱动程序提供一种简洁的接口。驱动程序只要将要读写的设备号、读写的初始地址和长度告诉控制器即可，剩下的都由控制器来做，比如总线申请等。

3.6.2 驱动器控制电路

应该将磁盘控制器和磁盘驱动器的控制电路区别开来，二者是作用于不同物理位置的。磁盘驱动器控制电路位于磁盘驱动器上，它专门负责直接驱动磁头臂做运动来读写数据：而主板上的磁盘控制器专门用来向磁盘驱动器的控制电路发送指令，从而控制磁盘驱动器读写数据。由磁盘控制器对磁盘驱动器发出指令，进而操作磁盘，CPU 做的仅仅是操作控制器就可以了。来梳理一下这个结构，CPU 通过主板上的导线发送指令给同样处于主板上的磁盘控制器，磁盘控制器继而通过线缆发送指令给磁盘驱动器，由磁盘驱动器来控制磁头臂。

CPU 操作控制器不需要多少复杂的指令，CPU 操作控制器的指令系统叫做磁盘控制器驱动程序。CPU 通过执行磁盘控制器驱动程序，生成指令发送给磁盘控制器，控制器收到这些指令后，通过电路逻辑运算生成另一种指令，也就是发送给磁盘驱动器的指令，这些指令就是常说的 ATA 指令集或者 SCSI 指令集。ATA 指令集是由 ATA 磁盘控制器发送给 IDE 磁盘驱动器的，而 SCSI 指令集是由 SCSI 磁盘控制器发送给 SCSI 磁盘驱动器的。

3.6.3 磁盘控制器驱动程序

那么机器刚通电，操作系统还没有启动起来并加载磁盘控制器驱动的时候，此时是怎么访问磁盘的呢？

CPU 必须执行磁盘控制器驱动程序才能产生将要发送给磁盘驱动器控制电路的指令，才能读写数据。不过，系统 BIOS 中存放了初始化系统所必需的基本代码。系统 BIOS 初始化过程中有这么一步，就是去查找磁盘控制器的 BIOS 地址，然后执行这个地址上的代码，也就是磁盘控制器的 BIOS 代码。来初始化磁盘控制器，并向系统 BIOS 报告控制器所掌管的磁盘设备的情况。最后 BIOS 让 CPU 发送指令，提取磁盘的 0 磁道的第一个扇区中的代码载入内存执行，从而加载 OS。

所以说系统 BIOS 中是包含基本的磁盘控制器驱动程序的，只不过功能不完善而已。在 OS 内核启动过程中，会用自己的完整的、全功能的磁盘控制器驱动来接管一开始被 CPU 执行的系统 BIOS 中的简化驱动。当然，BIOS 中也要包含键盘驱动，如果支持 USB 移动设备启动，还要有 USB 驱动。

图 3.30 显示了磁盘控制器驱动程序、磁盘控制器和磁盘驱动器控制电路三者的关系。

大家应该了解，如果要向 SCSI 磁盘上安装操作系统，则安装之前需要手动加载 SCSI 驱动，这样才能在安装程序的磁盘列表中找到正确的磁盘。因为 CPU 只有通过执行磁盘控制器驱动程序才能向磁盘驱动器发指令从而读取数据，如果此时 CPU 不知道去哪里执行驱

动程序，当然也就无法和磁盘控制器通信了，磁盘控制器也就无法发出 SCSI 或者 ATA 指令了。

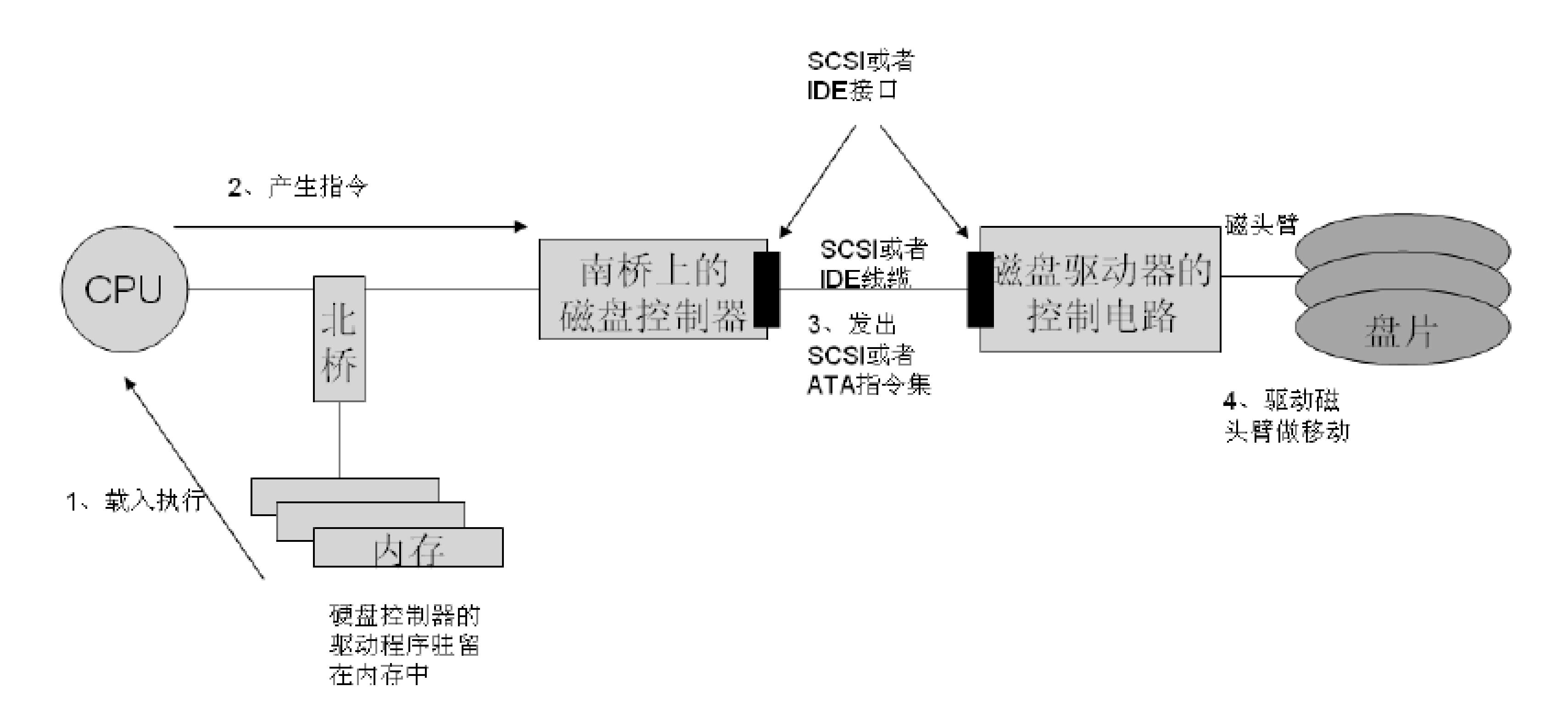


图3.30　磁盘控制器驱动程序、磁盘控制器和磁盘驱动器控制电路三者的关系

在启动操作系统之前，CPU 都是直接从 ROM 的地址中来执行磁盘控制器驱动程序的，直到驱动程序本身的数据从硬盘读入到内存之后。此时就会发生一个跳转，CPU 不再从 ROM 地址中提取驱动程序来执行，而是向 RAM 中的对应地址来提取驱动程序代码执行，此时 ROM 才真正发挥完它的作用。同样，如果从 USB 设备启动，OS 自身的 USB 驱动没有加载到 RAM 之前，也是由 ROM 来提供简化 USB 驱动的。

安装操作系统时，也可以从 ROM 中执行驱动程序代码，但是 ROM 的速度远远没有 RAM 快，由于安装操作系统主要是向磁盘写入数据，所以驱动程序只能一直从 ROM 中读取执行，这样就会变得很慢。如果操作系统已经安装好，启动之初从 ROM 中读取驱动程序执行也会很慢，但是一旦将 OS 本身的驱动程序读入内存并跳转到内存来执行，抛弃了 ROM，速度就立即增加了。而这一小段时间的慢速是可以接受的，而且是不得不接受的(总不能每次启动操作系统的时候都手动加载驱动吧)。

但是安装过程中需要在内存中没有驱动程序代码的情况下向磁盘复制大量数据的，这时候速度太慢是不能容忍的，所以必须手动加载驱动程序到内存执行，然后再进行安装。当然，也可以将 ROM 中的驱动程序代码复制到内存执行，但是 ROM 中的代码都是非常简化的通用版本，功能有限，只能用作操作系统启动这种临时任务，用于安装操作系统，性能是很差的。那么为何向 ATA 磁盘安装操作系统不需要加载驱动呢？这是因为 SCSI 卡的生产商很多，各自都有自己特别的功能，而 ATA 已经是非常成熟的技术，控制器一般都是 Intel 的，所以操作系统安装程序在初始化执行的时候，就已经自行加载了 ATA 驱动。



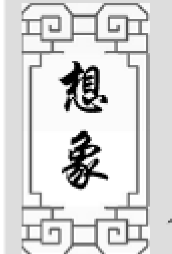
CPU 执行控制器驱动程序而发给控制器的信号，不是 ATA 或者 SCSI 指令集，指令集仅仅指的是控制器到硬盘驱动器之间的数据交互协议。而驱动程序所生成的信号只是告诉硬盘控制器去读或写总线上的哪个设备，地址段是多少，读出之后是否直接写入内存(DMA)等。而 SCSI 总线的仲裁等过程完全是靠控制器自己电路的逻辑来完成的，CPU 不需要参与仲裁运算，只管在任意时刻向控制器读写数据即可。

3.7 内部传输速率和外部传输速率

3.7.1 内部传输速率

磁盘的内部传输速率指的是磁头读写磁盘时的最高速率。这个速率不包括寻道以及等待扇区旋转到磁头下所耗费时间的影响。它是一种理想情况，即假设磁头读写的时候不需要换道，也不专门读取某个扇区，而是只在一个磁道上连续地循环读写这个磁道的所有扇区，此时的速率就叫做硬盘的内部传输速率。

通常，每秒 10000 转的 SCSI 硬盘的内部传输速率的数量级大概在 1000MB/s 左右。但是为何实际使用硬盘的时候，比如复制一个文件，其传输速率充其量只是每秒几十兆字节呢？原因就是磁头需要不断换道。



闪电侠正在做数学题，假设我们不打断他，他每秒能连续做 100 道数学题，此时我们每隔 0.1 秒，就和他交谈打断他一次，每次交谈的时间是 0.5 秒。也就是说闪电侠实际上做数学题的时间是每隔 0.5 秒做一次，每次只能做 0.1 秒的时间。这样，每 0.6 秒闪电侠只能做 10 道题，那么可以计算出闪电侠实际每秒能做的数学题只有区区 16 道，这和我们不打断他时的每秒 100 道题相比差了 6 倍多。

同样，磁盘也是这个道理，我们不断地用换道来打断磁头。磁头滑过盘片一圈，只需要很短的时间，而换道所需的时间远远比盘片旋转一圈耗费的时间多，所以造成磁盘整体外部传输速率显著下降。有人问，必须要换道么？如果要读写的数据仅仅在一条磁道上，那是可以获得极高的传输速率的，但是这个并不容易实现。如今，随着硬盘容量的加大，应用程序产生的文件更是在肆无忌惮地加大，动辄几十、几百兆甚至上 GB 大小，敢问这种文件用一个磁道能放下么？显然不能。

所以，磁头必须不断地被“打断”去进行换道操作，整体传输速率就会大大降低。实际中一块 10000 转的 SCSI 硬盘的实际外部传输速率也只有 80MB/s 左右。为了避免磁头被不断打断的问题，人们发明了 RAID 技术，让一个硬盘的磁头在换道时，另一个磁盘的磁头在读写。如果有很多磁盘联合起来，同一时刻总有某块硬盘的磁头在读写状态而不是都在换道状态，这就相当于一个大虚拟磁盘的磁头总是处于读写状态，所以 RAID 可以显著提升传输速率。不仅如此，如果我们将 RAID 阵列再次进行联合，就能将速率在 RAID 提速的基础上，再次成倍地增加。这种工作，就需要大型磁盘阵列设备来做了，磁盘阵列技术将在本书第 6 章介绍。



RAID 技术的细节在本书第 4 章详细阐述。

3.7.2 外部传输速率

磁头从盘片上将数据读出，然后存放到硬盘驱动器电路板上的缓存芯片内，再将数据从缓存内取出，通过外部接口传送给主板上的硬盘控制器。从外部接口传送给硬盘控制器时候的传输速率，就是硬盘的外部传输速率。这个动作是由硬盘的接口电路来发起和控制的。接口电路和磁头控制电路是不同的部分，磁头电路部分是超精密高成本的部件，保证磁头读写时候的高速率。但是因为磁头要被不断地打断，所以外部接口传输速率无须和磁头传输速率一样，只要满足最终的实际速率即可。外部接口的速率通常大于实际使用中磁头读写数据的速率(计算入换道的损失)。

3.8 并行传输和串行传输

3.8.1 并行传输

来举一个例子，有 8 个数字从 1 到 8，需要传送给对方。此时我们可以与对方连接 8 条线，每条线上传输一个字符，这就是并行传输。并行传输要求通信双方之间的距离足够短。因为如果距离很长，那么这 8 条线上的数字因为导线电阻不均衡以及其他各种原因的影响，最终到达对方的速度就会显现出差距，从而造成接收方必须等 8 条线上的所有数字都到达之后，才能发起下一轮传送。

并行传输应用到长距离的连接上就无优点可言了。首先，在长距离上使用多条线路要比使用一条单独线路昂贵；其次，长距离的传输要求较粗的导线，以便降低信号的衰减，这时要把它们捆到一条单独电缆里相当困难。IDE 硬盘所使用的 40 或者 80 芯电缆就是典型的并行传输。40 芯中有 32 芯是数据线，其他 8 芯是承载其他控制信号用的。所以，这种接口一次可以同时传输 32bit 的数据，也就是 4 字节。



IO 延迟与 Queue Depth

IO 延迟是指控制器将 IO 指令发出之后，直到 IO 完成的过程中所耗费的时间。目前业界有不成文的规定，只要 IO 延迟在 20ms 以内，此时 IO 的性能对于应用程序来说都是可以接受的，但是如果大于 20ms，应用程序的性能将会受到比较大的影响。

我们可以推算出，存储设备应当满足的最低 IOPS 要求应该为 1000/20 = 50，即只要存储设备能够提供每秒 50 次 IO，则就能够满足 IO 延迟小于等于 20ms 的要求。但是每秒 50 次，这显然太低估存储设备的能力了。

单块 SATA 硬盘能够提供最大两倍于这个最低标准的数值，而 FC 磁盘则可以达到 4 倍于这个数值。而对于大型磁盘阵列设备，由若干磁盘共同接受 IO，加上若干个 IO 通道并行工作，目前中高端设备达到十几万的 IOPS 已经不成问题。

然而，不能总以最低标准来要求存储设备。当接受 IO 很少的时候，IO 延迟一般会很小，比如 1ms 甚至小于 1ms。此时，每个 IO 通道的 IOPS = 1000/1 = 1000，这个数值显然也不对，上文所述的几十万 IOPS，如果每个 IO 通道仅提供 1000 的

IOPS，那么达到几十万，需要几百路 IO 通道，这显然不切实际。那么几十万的 IOPS 是怎么达到的呢？这就引出了另一个概念：Queue Depth。

控制器向存储设备发起的指令，不是一条一条顺序发送的，而是一批一批的发送，存储目标设备批量执行 IO，然后将数据和结果返回控制器。也就是说，只要存储设备肚量和消化能力足够，在 IO 比较少的时候，处理一条指令和同时处理多条指令将会耗费几乎相同的时间。控制器所发出的批量指令的最大条数，由控制器上的 Queue Depth 决定。如果连接外部独立磁盘阵列，则一般主机控制器端可以将其 Queue Depth 设置为 64、128 等值，视情况而定。

如果给出 Queue Depth、IOPS、IO 延迟三者中的任意两者，则可以推算出第三者，公式为：IOPS =（Queue Depth）×（IO Latency）。实际上，随着 Queue Depth 的增加，IO 延迟也会随即增加，二者是互相促进的关系，所以，随着 IO 数目的增多，将很快达到存储设备提供的最大 IOPS 处理能力，此时 IO 延迟将会陡峭的升高，而 IOPS 则增加缓慢。好的存储系统，其 IO 延迟的增加应该是越缓慢越好，也就是说存储设备内部应该具有快速 IO 消化能力。而对于消化不良的存储设备，其 IO 延迟将升高地很快，以致于在 IOPS 较低时，IO 延迟已经达到了 20ms 的可接受值。消化能力再高，也有饱和的时候。如图 3.31 所示。

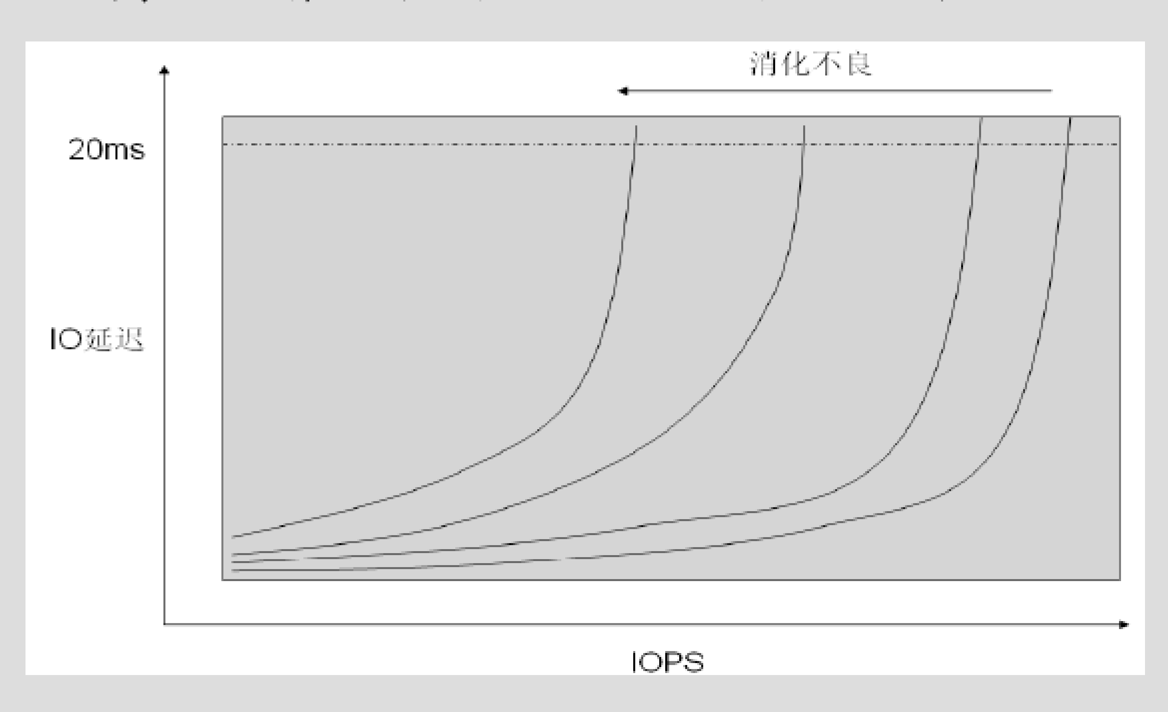


图 3.31 IO 延迟与 Queue Depth 示意图

3.8.2 串行传输

还是上面的例子，如果只用一条连线来连接到对方，则我们依次在这条线上发送这 8 个数字，需要发送 8 次才能将数字全部传送到对方。串行传输在效率上，显然比并行传输低得多。但是串行也有串行的优势，就是凭借这种优势使得硬盘的外部接口已经彻底被串行传输所占领。USB 接口、IEEE 1394 接口和 COM 接口，这些都是串行传输的计算机外部接口。

并行传输表面上看来比串行传输效率要高很多倍，但是并行传输有不可逾越的技术困难，那就是它的传输频率不可太高。由于在电路高速震荡的时候，数据线之间会产生很大的干扰，造成数据出错，所以必须增加屏蔽线。即使加了屏蔽线，也不能保证屏蔽掉更高的频率干扰。所以并行传输效率高但是速度慢。而串行传输则刚好相反，效率是最低的，每次只传输一位，但是它的速度非常高，现在已经可以达到 10Gb/s 的传输速率，但传输导线不能太多。

这样算来，串行传输反而比并行传输的总体速率更快。串行传输不仅仅用于远距离通信，现在就连 PCI 接口都转向了串行传输方式。PCIE 接口就是典型的串行传输方式，其单

条线路传输速率高达 2.5Gb/s，还可以在每个接口上将多条线路并行，从而将速率翻倍，比如 4X 的 PCIE 最高可达 16X，也就是说将 16 条 2.5Gb/s 的线路并行连接到对方。这仿佛又回到了并行时代，但是也只有在短距离传输上，比如主板上的各个部件之间，才能承受如此高速的并行连接，远距离传输是达不到的。

3.9 磁盘的 IOPS 和传输带宽(吞吐量)

3.9.1 IOPS

磁盘的 IOPS，也就是每秒能进行多少次 IO，每次 IO 根据写入数据的大小，这个值也不是固定的。如果在不频繁换道的情况下，每次 IO 都写入很大的一块连续数据，则此时每秒所做的 IO 次数是比较低的；如果磁头频繁换道，每次写入数据还比较大的话，此时 IOPS 应该是这块硬盘的最低数值了；如果在不频繁换道的条件下，每次写入最小的数据块，比如 512 字节，那么此时的 IOPS 将是最高值；如果使 IO 的 payload 长度为 0，只包含开销，这样形成的 IOPS 则为理论最大极限值。IOPS 随着上层应用的不同而有很大变化。



如何才算一次 IO 呢？这是很多人都没有弄清楚的问题，也是定义很混乱的一个问题。其根本原因就在于一次 IO 在系统路径的每个层次上都有自己的定义。整个系统是由一个一个的层次模块组合而成的，每个模块之间都有各自的接口，而在接口间流动的数据就是 IO。那么如何才算“一次”IO 呢？以下列举了各个层次上的“一次”IO 的定义。

- 应用程序向操作系统请求：“读取 C:\read.txt 到我的缓冲区”。操作系统读取后返回应用程序一个信号，这次 IO 就完成了。这就是应用程序做的一次 IO。
- 文件系统向磁盘控制器驱动程序请求：“读取从 LBA10000 开始的后 128 个扇区”，然后“请读取从 LBA50000 开始的后 64 扇区”，这就是文件系统向下做的两次 IO。这两次 IO，假设对应了第一步里那个应用程序的请求。
- 磁盘控制器驱动程序用信号来驱动磁盘控制器向磁盘发送 SCSI 指令和数据。对于 SCSI 协议来说，完成一次连续 LBA 地址扇区的读写就算一次 IO。但是为了完成这次读或者写，可能需要发送若干条 SCSI 指令帧。从最底层来看，每次向磁盘发送一个 SCSI 帧，就算一次 IO，这也是最细粒度的 IO。但是通常说磁盘 IO 都是指完成整个一次 SCSI 读或者写。
- 如果在文件系统和磁盘之间再插入一层卷管理层，或在磁盘控制器和磁盘之间再插入一层 RAID 虚拟化层，那么上层的一次 IO 就往往会演变成下层的多次 IO。
- 对于磁盘来说，每次 IO 就是指一次 SCSI 指令交互回合。一个回合中可能包含了若干 SCSI 指令，而这一个回合里却只能完成一次 IO，比如“读取从 LBA10000 开始的后 128 个扇区”。

例如，写入 10000 个大小为 1KB 的文件到硬盘上，耗费的时间要比写入一个 10MB 大小的文件多得多，虽然数据总量都是 10MB。因为写入 10000 个文件时，根据文件分布情况和大小情况，可能需要做好几万甚至十几万次 IO 才能完成。而写入一个 10MB 的大文件，如果这个文件在磁盘上是连续存放的，那么只需要几十个 IO 就可以完成。

对于写入 10000 个小文件的情况，因为每秒需要的 IO 非常高，所以此时如果用具有较高 IOPS 的磁盘，将会提速不少。而写入一个 10MB 文件的情况，就算用了有较高 IOPS 的硬盘来做，也不会有提升，因为只需要很少的 IO 就可以完成了，只有换用具有较大传输带宽的硬盘，才能体现出优势。

同一块磁盘在读写小块数据的时候是比较高的。而读写大块数据的时候是比较低的，因为读写花费的时间变长了。15000 转的硬盘比 10000 转的硬盘性能要高。图 3.32 所示为磁盘 IOPS 性能与 IO SIZE 的关系曲线。

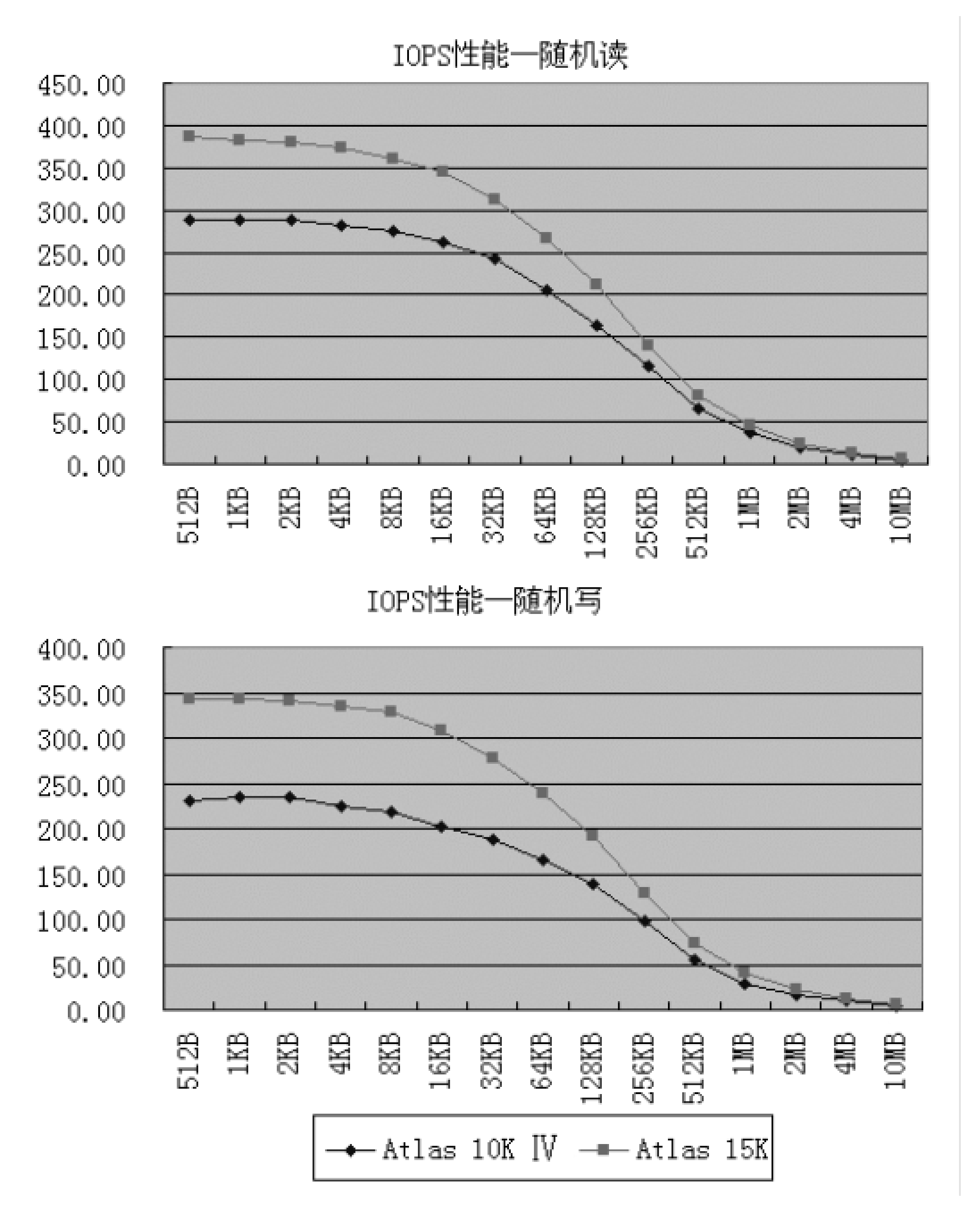


图3.32　磁盘 IOPS 性能与 IO SIZE 的关系曲线

3.9.2　传输带宽

传输带宽指的是硬盘或设备在传输数据的时候数据流的速度。还是刚才那个例子，如果写入 10000 个 1KB 的文件需要 10s，那么此时的传输带宽只能达到每秒 1MB，而写入一个 10MB 的文件用了 0.1s，那么此时的传输带宽就是 100MB/s。所以，即使同一块硬盘在写入不同大小的数据时，表现出来的带宽也是不同的。具有高带宽规格的硬盘在传输大块连续数据时具有优势，而具有高 IOPS 的硬盘在传输小块不连续的数据时具有优势。

同样，对于一些磁盘阵列来说，也有这两个规格。一些高端产品同时具备较高的 IOPS 和带宽，这样就可以保证在任何应用下都能表现出高性能。

3.10 小结：网中有网，网中之网

我们用图 3.33 来作为本章的结束。

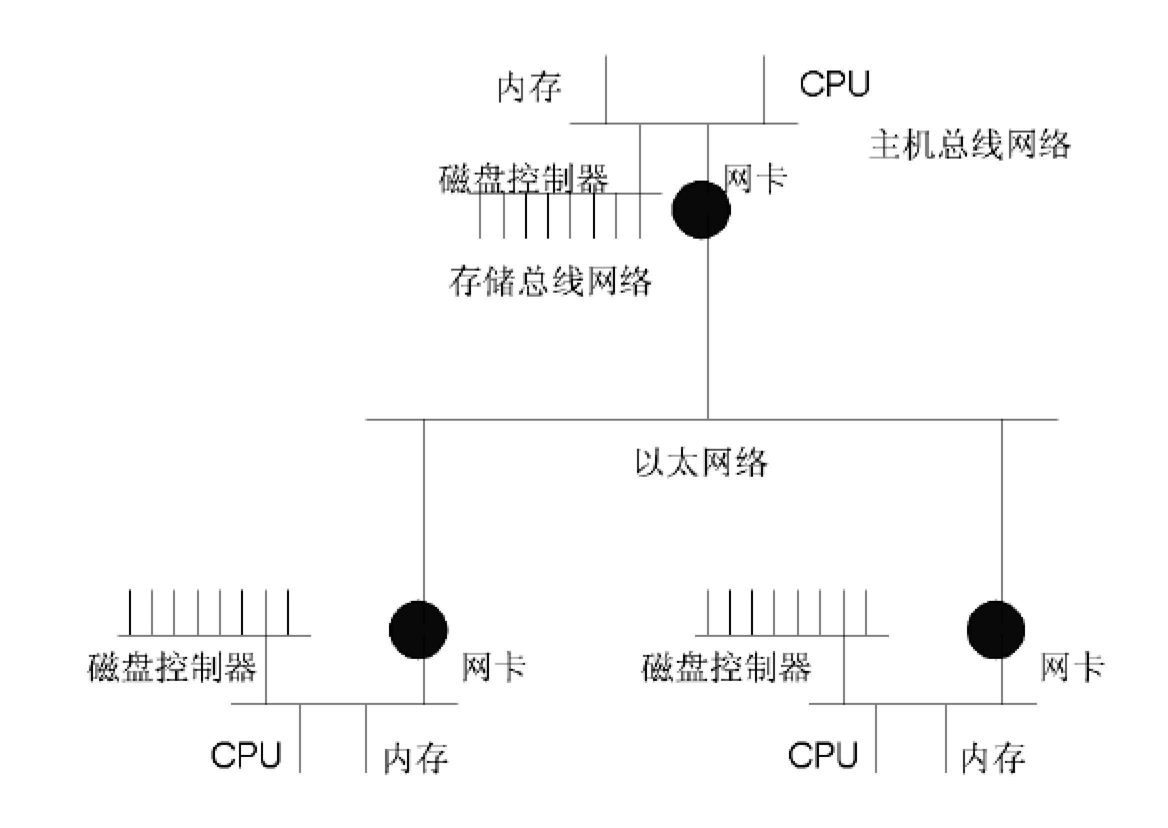


图3.33 三台计算机组成的网——网中有网



大话/详解七种 RAID

- RAID 1
- RAID 2
- RAID 3
- RAID 4
- RAID 5
- RAID 5E
- RAID 5EE
- RAID 6

第 3 章我们介绍了磁盘的内部原理、构造以及外部的接口系统。但是一块磁盘的容量是有限的，速度也是有限的。对一些应用来说，可能需要上百吉字节(GB)甚至几太字节(TB)大小的分区来存放数据。目前磁盘单块容量最多能到 1TB，这对于现代应用程序来说远远不够。

那么我们必须要制造单盘容量更大的硬盘么？为解决这个问题，人们发明了 RAID 技术，即 Redundant Array of Independent Disks 技术，中文意思是由独立的磁盘组成的具有冗余特性的阵列。既然是阵列，那一定需要很多磁盘来组成；既然是具有冗余特性的，那一定可以允许某块磁盘损坏之后，数据仍然可用。下面我们就来看一下 RAID 是怎样炼成的。

4.1 大话七种 RAID 武器

话说几千年前，有位双刀大侠，左右手各拿一把大刀。开始的时候，他总是单独使用每把刀，要么用左手刀，要么用右手刀。但是总被人打败，郁闷至极，于是苦心悟道，静心修炼。他逐渐模索出一套刀法，自称“合一刀法”，即双刀并用。外人看不见他的第二把刀，只能看到他拿着一把刀。他把两把刀的威力，合二为一，成为一把大刀！

而这种双刀合一的刀法，又可以分为两条路子，一条是常规路子，即这把合二为一的大刀，其实每出一招只有原来一把刀的威力，但是后劲更足了。一把刀顶不起来的时候，可以第二把刀上阵。对于敌人来说，还是只看见一把大刀。另一条是野路子，野路子往往效果很好，每出一招总是具有两把刀的威力之和，而且也具有两把刀的后劲！他实现这个野路子的方法，便是界内早有的思想——分而治之！也就是说他把一把刀又分成了很多细小的元素，每次出招时把两把刀的元素组合起来，所以不但威力大了，后劲也足了！

不过大侠的这个刀法有个致命的弱点就是双刀息息相关。一旦其中一把刀有所损坏，另一把刀相应的地方也跟着损坏。如果一把刀完全失去效力，那么另一把刀也跟着失效。

双刀大侠一直到临终也没有收一个徒弟，不是因为他武艺不精，而是因为他的合一刀法在当时被认为是野路子，歪门邪路的功夫，所以郁闷一生。临终时他用尽自己最后一点力气在纸上写下了 4 句诗后，抱撼而终！

刀于我手不为刀，
横分竖割成龙绦。
化作神龙游天际，
龙在我心任逍遥！

这就是后世流传的“合一刀谱”！俗称“龙谱”。

世上最高的刀法在心中，而不是手上！双刀大侠练就的是一门“浩瀚”绝学，一招一式都是铺天盖地，势不可挡！

几百年后，七星大侠在修炼了磁盘大挪移神功和龙谱之后的某一天，他突然两眼发愣：“朕悟到了！”然后奋笔疾书，成就了“七星北斗阵”这个空前绝后的阵式！ RAID 0 阵式就是这个阵式的第一个阵式！我们就来看看这个阵式的绝妙之处吧！

4.1.1 RAID 0 阵式

首先，这位七星大侠一定是对磁盘大挪移神功有很高的造诣，因为他熟知每块磁盘上面的磁性区域的构造，包括磁道、磁头、扇区和柱面等，这些口诀心法已经烂熟于心。在他看来，盘片就像一个蜂窝，上面的每一个孔都是一个扇区，可以说他已经参透了磁盘。其次，七星大侠一定是对合一刀法的精髓有很深的领悟，即他能领会双刀大侠那 4 句诗的含义，特别是第二句给了他很大的启发！“横分竖割成龙绦”，暗示着双刀大侠把他的刀在心中分割成了横条带和竖条带，所以叫“横分竖割”。分割完毕之后，双刀大侠把这些分割后所谓的“绦”，即细条带，在心中组合起来形成一条虚拟的“龙”，然后用龙来当

作武器，即“龙在我心任逍遥”。

这显然给了七星大侠很大的启发，何不把几块磁盘也给“横分竖割”，然后组成“龙”呢？对，就这么干！七星大侠卖血换来两块磁盘，找了个破庙，在后面搭了个草堆，成天摆弄他那两块用血换来的磁盘。白天出去要饭，晚上回来潜心钻研！他首先决定把两块磁盘都分割成条带，形成“绦”，可是该怎么分好呢？合一刀法的思想主要有两条路，一条是懒人做法，不想动脑子，即威力小，后劲足那种。另一种是需要动脑子算的，即威力足，后劲也足那种。

第一种怎么实现呢？七星大侠冥思苦想，却发现被误导了。因为第一种根本不需要做“绦”。双刀大侠的诗只是描述了威力巨大的第二种路子。所以三下五除二，七星大侠写出了 RAID 0 阵式中的第一个套路：累加式。也就是说，磁盘还是那些磁盘，什么都不动，也不用“横分竖割”。数据来了，先往第一块磁盘上写。等写满之后，再往第二块上写。然后将这两块磁盘在心中组成一条龙，这就符合了合一刀法的思想。只不过这条龙威力比较弱，因为每次 IO 只用到了一块磁盘，另一块磁盘没有动作。但是这条龙的后劲，比单个磁盘足了，因为容量相对一块磁盘来说，增大了。

第一套路子实现了，可第二套路子就难了。其实磁盘已经被分割了，扇区不就是被分割的么？但是一个扇区只有 512 字节大小，这不符合合一刀法。因为合一刀法中是“绦”而不是“粒”，分割成粒的话，不仅开销太大，而且组合起来也很困难。所以七星决定完全照搬合一刀法的思想，但是又不能丢弃磁盘已经分割好的扇区，所以七星绞尽脑汁想出这么一个办法，如图 4.1 和图 4.2 所示。

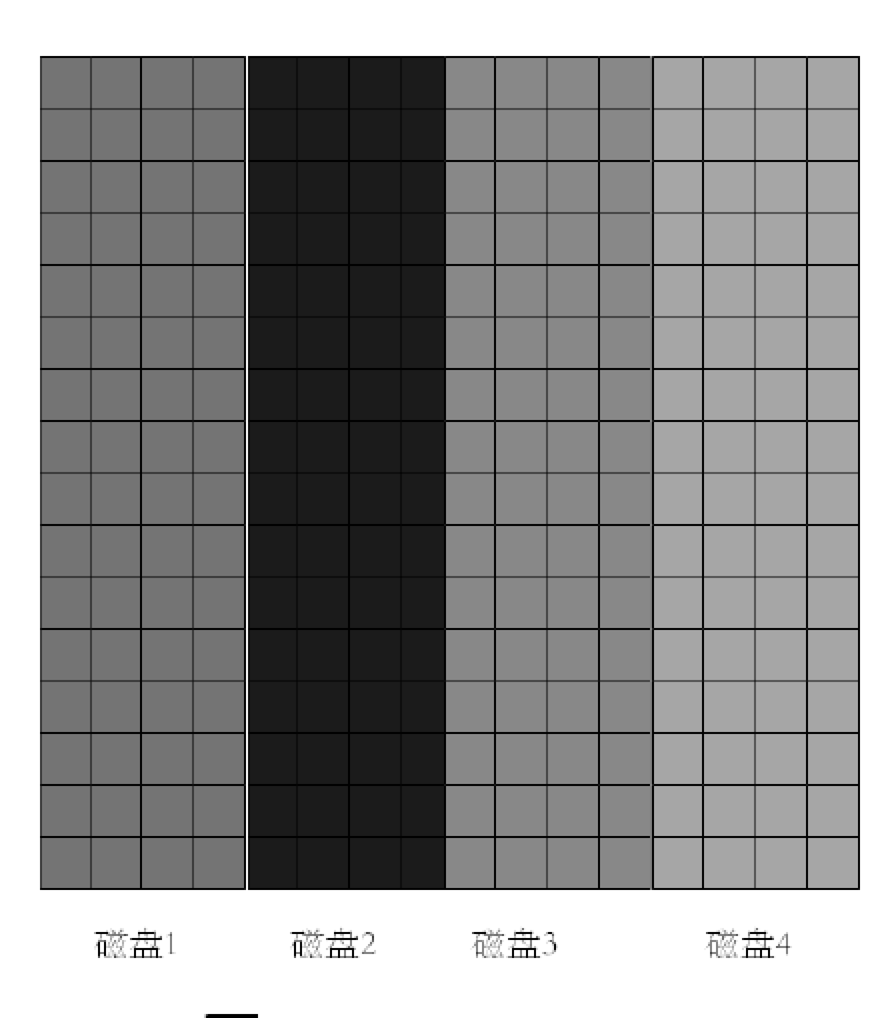


图4.1 正常的 4 块硬盘

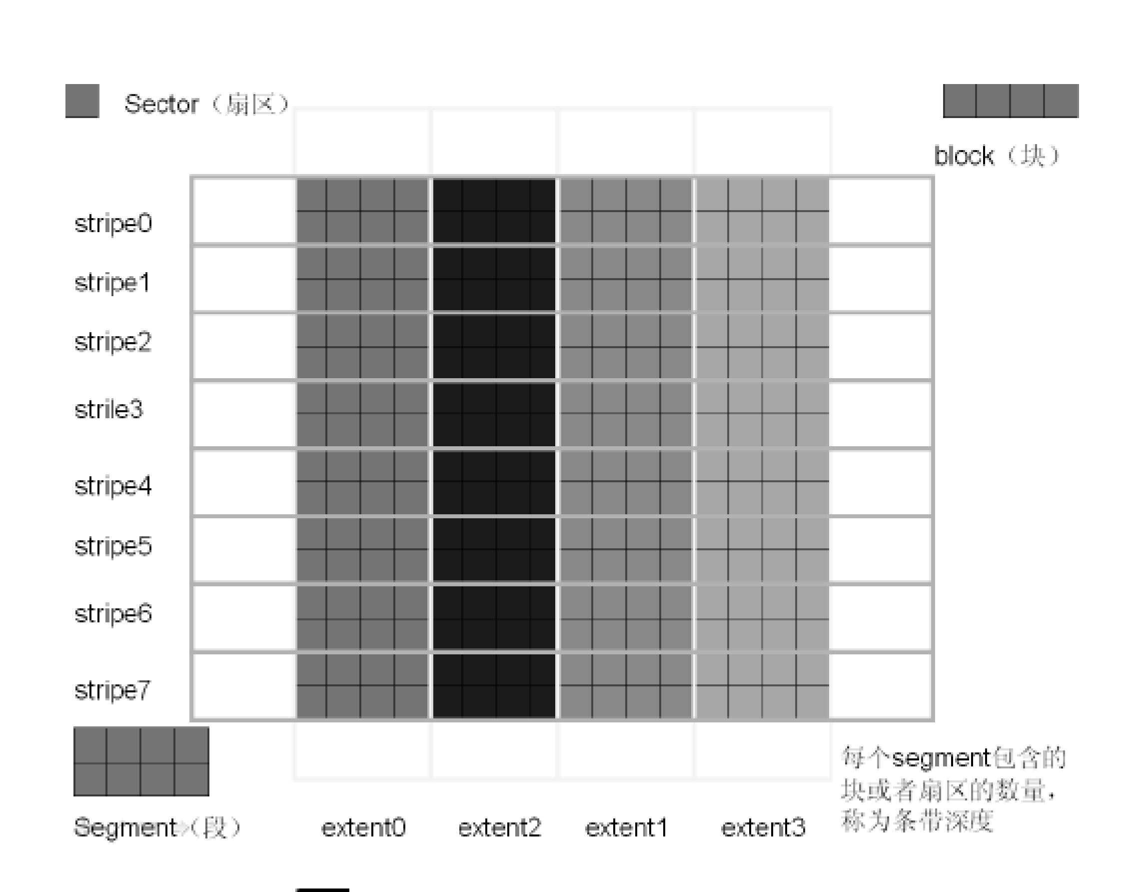


图4.2 被分割的 4 块硬盘

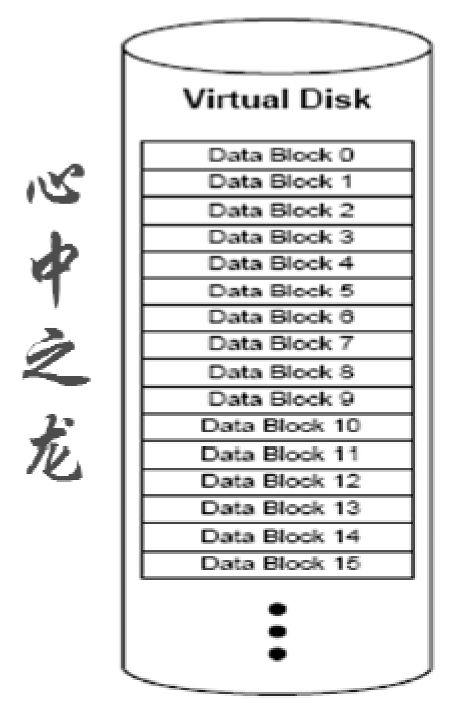
图 4.1 所示的是 4 块普通硬盘，其上布满了扇区。扇区是实实在在存在于盘片上的，具有自己的格式。图 4.2 所示的是引入分割思想之后的硬盘。由于许多文件系统或者卷管理软件都使用块而不是扇区作为基本存储单元，所以图 4.2 中也使用由 4 个扇区组成的块作为基本单元。不同磁盘的相同偏移处的块组合成 Stripe，也就是条带。

块的编号也是以横向条带方向开始一条一条的向下。这样，对于一个全新的文件系统和 RAID 0 磁盘组，如果有大块数据写入时，则数据在很大几率上可以以条带为单位写入。

也就是说数据被分成多块写入 4 个硬盘，而不是单硬盘系统中的顺序写入一个硬盘，这就大大提高了速度。图 4.3 所示的为多块磁盘组成的逻辑磁盘示意图。

提示

磁盘上实实在在存在的只有扇区结构，Stripe 并不是一个实实在在的结构，它只是由程序根据算法公式现套现用的。就像戴了一副有格子的眼镜看一张白纸，那么会认为这张白纸被格式化了，其实并没有。另外，条带化之后的多块硬盘，数据是被并行写入所有磁盘的，也就是多管齐下，而不是横向写满一个条带，再写下一个条带。



七星生成他的心中之龙，也有两套路子，第一套路子，也就是不分割，只把两块磁盘在心中合并成一块，第一块用完了，接着用第二块。在外看来，就是一整块大磁盘。
第二套路子，也就是分割成“绦”，然后两块磁盘轮流上阵，你出一招，我出一式，你再出一招，我再出一式，也就是 extent()在磁盘 1，extent3 在磁盘 2，extent3 又回到磁盘 1。当然如果多块盘，那么 extent 就轮流转，就这样组成心中之龙。

图4.3 心中之龙

七星大侠就这样埋头苦苦思考了整整 1 年，基于合一刀法的横分竖割的思想，完成了“七星北斗阵”的第一个阵式——RAID 0 阵式。

4.1.2 RAID 1 阵式

花开七朵，各表其一。话说七星在完成 RAID 0 阵式之后，并没有沾沾自喜，而总是想在合一刀法上有所创新。



RAID 0 阵式纵然威力无比，但弱点也很明显，一旦其中一块磁盘废掉，整个阵式将会被轻易攻破。因为每次出招靠的就是“合一”，如果任意一块坏掉，也就没有“合”的意义了。也就是说，数据被我在心中分割，本来老老实实写到一块盘就完事了，可为了追求威力，非要并发写盘，第一、三、五……块数据写到了 1 号盘，第二、四、六……块数据写到了 2 号盘。但是对于外界来说，会认为是把数据都写到了我心中的一块虚拟盘上。这样不坏则已，一旦其中任何一块磁盘损坏，就会数据全毁，因为数据是被分割开存放在所有磁盘上的。不行，太不保险了。为了追求威力，冒的险太大，要想个稳妥的办法。

于是七星再次冥思苦想，终于创出了 RAID 1 阵式！

这话要说到 800 年前，有位“独行侠”，心独身独终日孤单一人。据称他每次出招从来不用双手，总是单手打出单掌，练就了一门“独孤影子掌”。虽说此掌法威力不高，但

是自有其妙处。每当他敌不过他人，单掌被击溃的时候，就会立即换用另一只从来都没用过的掌继续出招。这一绝学往往另自以为已经占了上风的敌人在还没有回过神来的情况下，被打个落花流水！不但他的掌法绝妙，就连他的身法都达到了炉火纯青的地步。他能修炼出一个影子，这个影子平时总是跟随着他，他做什么，影子就做什么。一旦真身损毁，其影子便代替他的真身来动作。这位独行侠遗留的“孤独影子掌”秘笈如下。

心朦胧，掌朦胧。
掌由独心生。
身朦胧，影朦胧。
身影心相同。
花朦胧，夜朦胧。
独饮赏月容。
灯朦胧，人朦胧。
此景何时休？

独行侠的这段诗句不难理解，孤独给了他灵感，身独心也独，如此练就的功夫，也是独孤残影。最后一句说出了大侠的无奈，其实他也不想孤独，但是没人能理解他。

七星大侠领悟了独行侠的苦衷，感受了他的心境。独行侠练就的是一门“无奈”绝学，处处体现着凄惨与潦迫。只有残了，才能重获新生。一只掌断了，另一只掌才能接替，这是何等凄惨？简直凄惨至极！不过往往孤独凄苦的人都很注意自保，虽然招式的威力是最小的，但是这门学问是武林中用于自保的最佳选择。

七星大侠没有理由不选择这门自保神功来解决他在 RAID 0 阵式中的破绽，也就是安全问题。毫无疑问，RAID 0 是强大的，但是也是脆弱的，一点点挫折就足以让 RAID 0 解体。

七星大侠决定完全抛弃 RAID 0 的思想，采用独行侠的思想。将两块磁盘中的一块用于正常使用，另一块用作正常使用磁盘的影子。影子总是跟随主人，主人做什么，影子就做什么。工作盘写了一个数据，影子盘在相同位置也写上了数据。读数据的时候，因为数据有两份，可以在工作盘读，也可以到影子盘读，所以增加了并发性。即修炼这个阵式的人，可以同时应付 2 个敌人的挑衅，自身应付一个，影子应付一个，这无疑是很高明之处！

但是应付一个敌人的时候，不像 RAID 0 阵式那样可以同时使用多块磁盘，只能使用一块磁盘。当其中一块磁盘坏掉，或者其中一块磁盘上某个区域坏掉，那么对应影子盘或者影子盘上对应的位置便会立即接替工作盘，敌人看不出变化。可能独行侠一生都没有遇到同时和两个对手过招的情况，所以在他的秘笈中，并没有体现“并发读”这个功能，只体现了安全自保。

然而七星并没有全面抛弃双刀大侠的思想，而是保留了双刀的精华，即“横分竖割”的基本思想，抛弃了他的算法，即鲁莽而不计后果的并发往各个磁盘上写数据的方法。所谓算法，也即指大侠对付敌人招数的时候，在心中盘算的过程。心算的速度远远快于出招的速度，所以心算引发的延时并不会影响出招。现在江湖人士也大多都是精于钻研算法，而只有制造兵器的铁匠才去钻研如何用料，考虑如何才能减轻兵器重量而不影响兵器的硬度和耐磨度等。

可以说，兵器的材质、设计加上大侠们精心的算法才形成了江湖上形形色色的功夫秘

笼！而材质在很大程度上发展是很慢的，想有突破非常困难。但是算法就不同了，大侠可以研究出各种使用兵器的方法，将兵器用的神乎其神！磁盘的转速、磁密度、电路等，虽然一直在提升，但是终究太慢。所以出现了以七星大侠为代表的算法派，他们苦研算法，用来提高磁盘的整体性能。假想某天一旦某个铁匠造出了屠龙刀倚天剑这般的神器，我想七星这等大侠也就无用武之处了。可惜这两把神器已经在自相残杀中玉石俱焚了。

七星大侠最后给这个阵式起名叫做“RAID 1”。图 4.4 显示了 RAID 1 组成的逻辑磁盘。

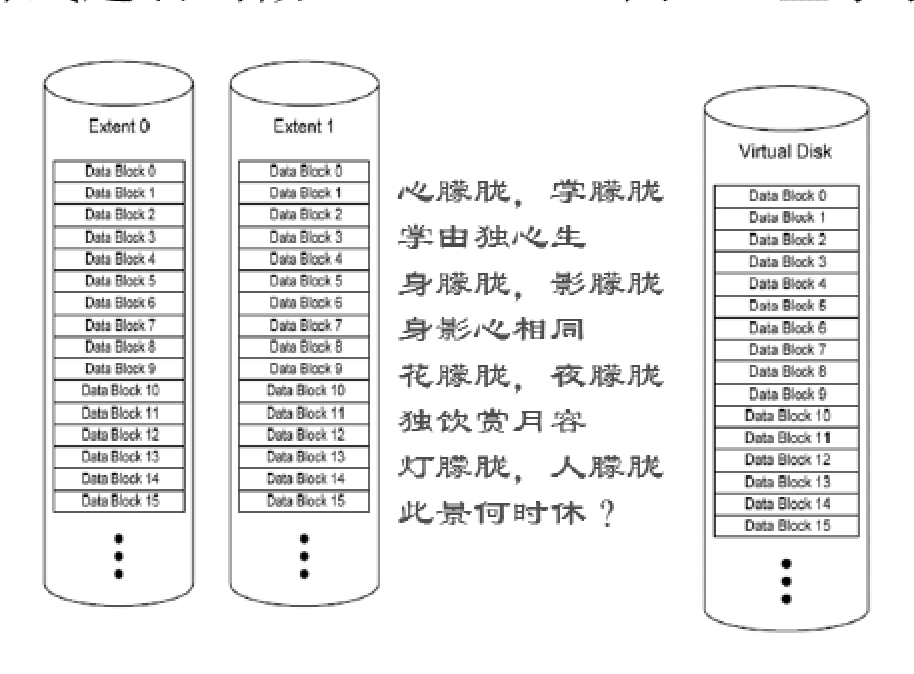


图4.4 RAID 1 示意图

但是七星也深深认识到，这个 RAID 1 阵式还是有两个大弱点。

- 在修炼的时候，速度会稍慢，因为每次修炼，除了练真身之外，还要练影子。不然影子不会，出招的时候影子就无法使用。这会对实际使用有一定影响，数据写到工作盘上，也必须写到影子盘上一份。
- 虽然自己有个影子，但是影子没有给真身增加后劲。真身累了，影子也会被拖累。不管修炼了几个影子，整体的耐力和体力只等于其中一个的体力耐力，也就是真身的体力和耐力。整体的体力和耐力都被限制在体力耐力最小的影子盘或者真身上。也就是说 RAID 1 提供的最大容量等于所有组成 RAID 1 的磁盘中，容量最小的一块，剩余容量不被使用，RAID 1 磁盘组的写性能等于所有磁盘中性能最低的那块磁盘的写性能。

七星看了看 RAID 0，又看了看 RAID 1，一个鲁莽急躁但威力无比，一个独孤残苦自嘲自保。矣！呜呼哉！！七星心想，我怎么走了两个极端呢？不妥，不妥，二者皆不合我意！于是，七星大侠又开始了苦心钻研，这一下就是 2 年！

4.1.3 RAID 2 阵式

提示

史话：话说明末清初时期，社会动荡，英雄辈出。有这么一位英雄，号称“优雅剑侠”，他持双手剑，得益于流传甚广的合一刀法，并加入了自己的招式，修炼成了一套“合一优雅剑法”。剑侠深知合一刀法的鲁莽招式，虽然威力巨大，但必会造成大祸。所以他潜心研究，终于找到一种办法，可以避免合一的鲁莽造成的不可挽回的祸患。他分析过，合一之所以鲁莽就是因为他没有备份措施，兵器有任何一点损坏都会一损具损。那么是不是可以找一种方法，对兵器上的每个条带都做一个备份，就像当年独行侠那样，但是又不能一个对一个，那就和独行侠无异了。

剑侠的脑子很好用，他从小精于算术，有常人不及之算术功夫。如今他终于发挥出他的算术技能了，他先找来一张纸，然后把他的两把剑和这张纸，并排摆在地上，然后对剑和纸进行横分竖割，然后一一对照，将第一把剑的第一格写上一个 1，然后在第二把剑的相同位置上写上一个 0，然后在纸的对应位置上算出前二者的和，即 1+0=1。然后剑侠设想，一旦第一把剑被损坏，现在只剩第二把剑和那张记满数字的纸，剑侠恍然大悟！原来如此精妙！

为什么呢？虽然第一把剑损坏，但是此时仍然可以出招，因为第一把剑上的数字可以用纸上对应位置的数字，减去第二把剑对应位置上的数据！也就是 1-0=1，选择就可以得出第一把剑上已经丢失的数字！而在敌人看来，仍旧是手持一把大剑，只不过威力变小了，因为每次出招都要计算一次。而且修炼的时候也更加难了，因为每练一招就要在纸上记录下双剑之和，而且还需要用脑子算，速度比合一刀法慢了不少。

哇哈哈，剑侠仰天长笑！他给自己的剑法取名“优雅合一剑法”，意即他的剑法比合一刀法虽然威力不及，但也差不多少。最重要的是，他克服了合一刀法鲁莽不计后果的弱点，所以要比合一刀法来的优雅。但是这个剑法也有弱点，就是他额外增加了一张纸和用了更多的脑筋来计算。脑子计算倒是不成问题，努力学习算法便可，但是额外增加了一张纸，这个难免有些遗憾，但是也没有办法，总比独行侠那一套自保好得多。自保的代价是修炼一个平时几乎用不到的影子，是一比一。优雅合一剑法是二比一，降低了修炼的代价，而威力却较合一刀法没减多少。

然而，这套剑法虽然声名大噪，但是优雅剑侠还是被一个突如其来的问题，一直折磨着，直到临终也没有想出办法解决。

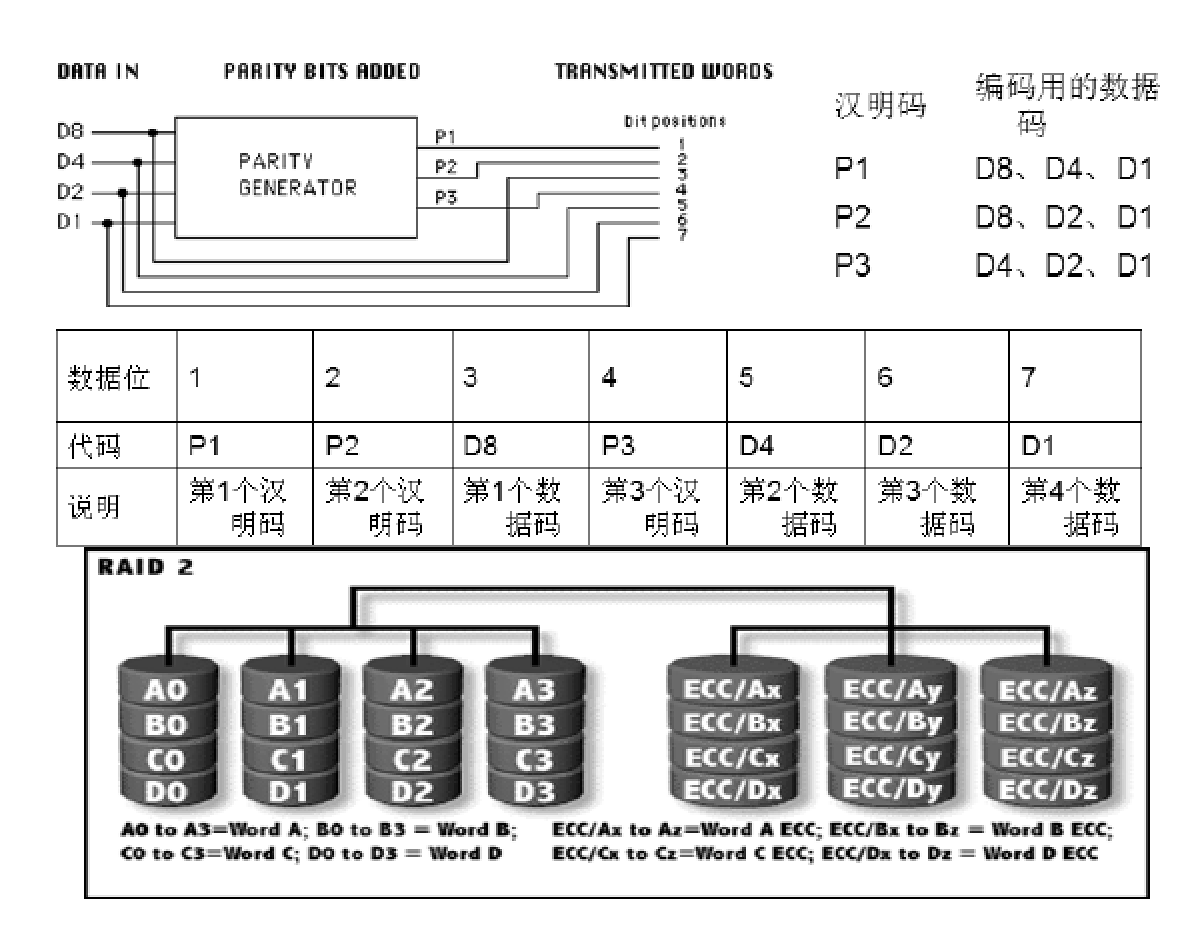
疑问 如果我使用 3 把剑、4 把剑、5 把剑，这套剑法还奏效么？因为 3 把剑的数字之和，就不是一个数字，而是两个数字了，比如 1 + 0 + 1 = 10。而这套剑法只有一张纸，一个格不能放两个数字，这样就必须再加一张纸，这样不就和独行侠那一套无异了么？比例太高，不妥。所以优雅剑侠一直再考虑这个问题，临终前留下一段诗句，也抱憾而终。

独行合一皆非道，
二者中庸方优雅。
加减算术勤思考，
世间正道为算法！

优雅剑侠这段诗的最后一句，指明了后人若要解决这个问题，必须要找到一种算法。不管多少个数字，如果掩盖一个数字，可以将其他数字代入这个算法，就可以得到被掩盖的数字的值。这在当时简直就是不可能的事。“世间正道为算法”这句话后来被作为推动武林发展的一句至理名言。随着近代西方科技传入中国，这种算法终于被人了解了！他是如此简单而美丽！他改变了整个世界！

峰回路转，七星大侠在优雅剑法的基础上，把剑换成了磁盘。那张记录数字之和的纸，七星也改用磁盘来记录。这样，组成了一个三磁盘系统，两块数据盘，一块所谓“校验盘”。当数据损坏时，根据校验盘上的数字，可恢复损坏磁盘上的数字。2 磁盘系统每次只能传输 2 路数据，因为数据盘就两块，而每块磁盘每次就传输出去一路。

RAID 2 的具体实现如图 4.5 所示。



汉明码	编码用的数据码
P1	D8、D4、D1
P2	D8、D2、D1
P3	D4、D2、D1

数据位	1	2	3	4	5	6	7
代码	P1	P2	D8	P3	D4	D2	D1
说明	第1个汉明码	第2个汉明码	第1个数据码	第3个汉明码	第2个数据码	第3个数据码	第4个数据码

图4.5　RAID 2 的具体实现

看看 RAID 2 具体的实现。值得研究的是，七星大侠并没有使用加减法来进行校验，而是用了一种算法复杂的所谓“汉明码”来校验。这可不是信手拈来，而是有一定原因的。用加减算法进行校验，并没有对数据纠错的能力。加减法情况下，比如 1+0+1=10，这段数据在从磁盘被传输给控制器的时候，校验位会一同传输，即数据位 101，校验位 10，此时经过控制器的校验，他会算出 1+0+1=10，和一同传过来的校验位进行比对。如果相同，则证明数据都无误。但是此时如果在传输的过程中，电路受到干扰，数据位其中有一位畸变了，比如变成 111 了，也就是中间那个 0 变成 1 了，其他不变，此时控制器进行计算 1+1+1=11，和一同传过来的 10 不同，那么控制器会怎么认为？他可以认为数据位全部正确，而校验位被畸变，也可以认为数据位被畸变，校验位正确。所以根本不能判断到底是哪种情况，所以不能修复错误。而七星引入了汉明码，汉明码的设计使得接收方可以判断到底是哪一位出错了，并且能修正一位错误，但是如果有两位都错了，那么就不能修正了。

Hamming Code ECC(汉明码错误检测与修正)

RAID 2 算法的复杂性在于它使用了很早期的纠错技术——汉明码(Hamming Code)校验技术。现在就来看一下汉明码的算法。

汉明码在原有数据位中插入一定数量的校验位来进行错误检测和纠错。比如，对于一组 4 位数据编码为例，汉明码会在这 4 位中加入 3 个校验位，从而使得实际传输的数据位达到 7 位，它们的位置如图 4.5 所示。

需要被插入的汉明码的位数与数据位的数量之间的关系为 $2^P \geqslant P+D+1$，其中 P 代表汉明码的个数，D 代表数据位的个数。比如 4 位数据，加上 1 就是 5，而能大于 5 的 2 的幂数就是 3($2^3=8$，$2^2=4$)。所以，7 位数据时需要 4 位汉明码($2^4>4+7+1$)，64 位数据时就要 7 位汉明码($2^7>64+7+1$)。

在 RAID 2 中，每个 IO 下发的数据被以位为单位平均打散在所有数据盘上。如图 4.5 中所示，左边的为数据盘阵列，如果某时刻有一个 4KB 的 IO 下发给这个 RAID 2 系统，则这 4KB 中的第 1、5、9、13 等位将被存放在第一块数据盘的一个扇区中，第 2、6、10、14 等位被存放在第二个磁盘的对应条带上的扇区，以此类推。这样，数据强行打散在所有磁盘上，迫使每次 IO 都要全组联动来存取，所以此时要求各个磁盘主轴同步，才能达到最佳效果。因为如果某时刻只读出了一个 IO 的某些扇区，另一些扇区还没有读出，那么先读出

来的数据都要等待，这就造成了瓶颈。主轴同步之后，每块磁盘盘片旋转同步，某一时刻每块磁盘都旋转到同一个扇区偏移的上方。同理，右边的阵列(我们称之为校验阵列)则是存储相应的汉明码。

RAID 2 在写入数据块的同时还要计算出它们的汉明码并写入校验阵列，读取时也要对数据即时地进行校验，最后再发向系统。通过上文的介绍，我们知道汉明码只能纠正一个位的错误，所以 RAID 2 也只能允许一个硬盘出问题，如果两个或以上的硬盘出问题，RAID 2 的数据就将受到破坏。

RAID 2 是早期为了能进行即时的数据校验而研制的一种技术(这在当时的 RAID 0、1 等级中是无法做到的)，从它的设计上看也是主要为了即时校验以保证数据安全，针对了当时对数据即时安全性非常敏感的领域，如服务器、金融服务等。但由于校验盘数量太多、开销太大及成本昂贵，目前已基本不再使用，转而以更高级的即时检验 RAID 所代替，如 RAID 3、5 等。

七星大侠现在已经创造了三种阵式了，根据合一刀法所创的 RAID 0，根据独孤掌所创的 RAID 1，根据优雅合一剑法所创的 RAID 2。而七星的郁闷之处和当年优雅剑侠一样，就是苦于找不到一种一劳永逸的绝妙算法，一种集各种优点为一身而且开销小的算法。

4.1.4 RAID 3 阵式

话说到了清末，清政府开展洋务运动，师夷长技以治夷。还别说，真引入了不少好技术，比如布尔逻辑运算式。这话要从布尔说起，布尔有一次在家捣鼓继电器，他将多个继电器时而串联，时而并联，时而串并一同使用，逐渐摸索出一些规律。比如两个继电器在串联时候，必须同时闭合两个开关，电路才能接通，灯泡才能亮。如果把开关闭合当作 1，开关关闭当作 0，灯泡点亮当作 1，灯泡不亮当作 0，那么这种串联电路的逻辑就可以这样写：1 和 1=1。也就是两个开关都闭合，灯泡才能亮(等于 1)。

然后，他还发现一个逻辑，如果在这个串联电路上增加一个元件，如果两个开关都闭合的时候，电路反而是断开的。有人说不可能，那么请仔细想一想，闭合开关电路断开，这有什么难的么？完全可以通过继电器来实现，比如电路闭合之后，电磁铁通电，把铁片吸引下来，而这个铁片是另一个电路的开关，铁片下来了，另一个电路也就断开了，所以通过把这两个电路组合，完全可以得到这种逻辑：1 和 1=0。

还有一种逻辑，就是当两个开关任意一个闭合时，电路就通路，也就是并联电路。这种逻辑可以这么表达：1 或 1=1、1 或 0=1。经过多种组合，布尔得到了 1 或 1=1、1 或 0=1、1 和 1=1、1 和 0=0。

这就是 4 种基本逻辑电路。这种“和”、“或”的运算，很多人都不理解。人们理解的只是加减运算，因为加减很常用。人们不理解的原因就是不知道除了加减算术之外，还有一种叫做“逻辑”的东西，也就是因果的运算。人们往往把 1 当成数量，代表 1 个，而在因果率中，1 不代表数量，它只代表真假，其实我们完全可以不用 1 这个符号来代表真，我们就用中文“真”代表真，可否？

当然可以，但是因为笔画太多，不方便，还是用 1 和 0 代表真假比较方便。其实磁盘上的数据，也不是 1 就代表 1 个，而是 1 代表磁性的取向，因为磁性只有两个取向，仿佛对称就是组成宇宙的基石一样，比如正负，对错等。当因果率被用数学方程式表达出来并

赋予电路的物理意义之后，整个世界也就进入了新世纪的黎明，这个世纪是计算机的世纪。

> **提示** 从数学到物理意义，我们仿佛看出点什么来，现代量子力学那一大堆数学式，折服了太多的科学家，包括爱因斯坦，到他去世前，爱因斯坦都没有理解量子力学所推演出来的数学式子，在物理上到底代表了什么意义。而且直到21世纪，也没有人给予这些式子以“目前”可理解的物理意义。我们可以想象一下布尔逻辑算式公布的时候，物理意义到底是什么？没人知道，甚至布尔自己估计也不知道，就只是一对式子而已。直到有一天一个人在家捣鼓继电器，突然风马牛不相及的想到了，这不就是布尔逻辑么？从此，数字电路，计算机时代，改变了我们的世界。

七星在学习了布尔逻辑算式之后，也是稀里糊涂地把它用在RAID 2那一直困扰他的问题上面，看看能否有所突破。

布尔运算中有一个XOR运算，即1 XOR 0=1，1 XOR 1=0，0 XOR 0=0。布尔也总结出了类似加法结合率，加法交换率等类似的逻辑运算率，并发现了一些规律。

```
1 XOR 0 XOR 1=0
0 XOR 1 XOR 0=1
```

假如第一个式子中，中间的 0 被掩盖，完全可以从结果推出这个被掩盖的逻辑数字。不管多少位，进行逻辑运算之后还是一位。仔细一想也是理所当然的，逻辑结果只有两个值，不是真，就是假，当然只用一位就可以代替了。

> **提示** 大家可以自己算算，不管等式左边有多少位进行运算，这个规律都适用。但是在加减法中，若要保持等式右边有一位，则左边参与运算的只能是1＋0或者0＋1，再多一个数的话，右边就是两位了。但是逻辑运算中等式右边永远都是一位！就是如此绝妙。为什么如此精妙呢？没人能解释为什么，就像问为什么有正电荷，负电荷一样，他们到底是什么东西，谁也说不清。

七星大侠开始并不觉得这是真的，他反复演算，想举出一个伪证，可是徒劳无功。七星不得不为布尔的绝学所折服！同时也为西方发达的基础科学所赞叹！

至此，困扰七星大侠多年的关于算法的问题，终于随着西方科学的介入，得以顺利解决！解决地是那么完美，那么畅快！

七星立即决定投入其下一个阵式RAID 3的创立过程中。他发狂似的抛弃了那冗余的让人看着就不顺眼的RAID 2的几块校验盘，只留下一块。按照布尔的思想，数据盘的每一个位之间做XOR运算，然后将结果写入校验盘的对应位置。这样，任何一块数据盘损坏，或者其中的任何一个扇区损坏，都可以通过剩余的位和校验位一同进行XOR运算，而运算的结果就是这个丢失的位。8位一起校验可以找出一个丢失的字节，512字节一起校验就可以找到一个丢失的扇区。

做到这里，已经算是成功了，但是七星还不太满足，因为他还有一桩心事，就是RAID 2中数据被打得太散了。七星大侠索性把RAID 3的条带长度设置成为4K字节，这样刚好适配了上层的数据组织，一般文件系统常用的是以4KB为一个块。如果用4块数据盘，则条带深度为2个扇区或者1KB。如果用8个数据盘，则条带深度为1个扇区或者说512字节。

总之，要保持条带长度为上层块的大小。上层的 IO 一般都会以块为单位，这样就可以保证在连续写的情况下，可以以条带为单位写入，大大提高磁盘并行度。

七星在 RAID 3 阵式中，仍旧保持 RAID 2 的思想，也就是对一个 IO 尽量做到能够分割成小块，让每个磁盘都得到存放这些小块的机会。这样多磁盘同时工作，性能高。所以七星在 RAID 3 中把一个条带做成 4KB 这个魔术值，这样每次 IO 就会牵动所有磁盘并行读写。到此我们了解了，RAID 2 和 RAID 3 都是每次只能做一次 IO(在 IO 块大于 Block SIZE 的时候)，不适合于要求多 IO 并发的情况，因为会造成 IO 等待。RAID 3 的并发只是一次 IO 的多磁盘并发存取，而不是指多个 IO 的并发。所以和 RAID 2 一样，适合 IO 块大，IO SIZE/IO PS 比值大的情况。



在极端优化的条件下，RAID 3 也是可以做到 IO 并发的。控制器向一块磁盘发送的读写指令，其中包含一个所要读取扇区的长度，如果下一次 IO 与本次 IO 在物理上是连续的(连续 IO)，此时如果控制器做了极端的优化，则可以将这两次 IO 合并起来，向磁盘发送的每个 IO 指令中包含了两次上层 IO 的数据，这样也算是一种并发 IO。当然，这种优化不仅仅可以在磁盘控制器这一层实现，其实文件系统层也可以实现。

提示

RAID 3 和 RAID 2 一样，要达到 RAID 3 的最佳性能，需要所有磁盘的主轴同步，也就是说。对于一块数据，所有磁盘最好同时旋转到这个数据所在的位置，然后所有磁盘同步读出来。不然，一旦有磁盘和其他磁盘不同步，就会造成等待，所以只有主轴同步才能发挥最大性能。

总结一下，RAID 3 相比 RAID 2 校验效率提升，成本减少(使用磁盘更少了)。缺点是不支持错误纠正了，因为 XOR 算法无法纠正错误。但是这个缺点已经不重要，发生错误的机会少之又少，可以完全靠上层来处理错误了。正可谓：

与非异或同，
一语解千愁。
今朝有酒醉，
看我数风流！

关于 RAID 3 的校验盘有没有瓶颈的问题

不妨用一个例子来深入理解一下 RAID 3。通过刚才的讲解，大家知道了 RAID 3 每次 IO 都会分散到所有盘。因为 RAID 3 把一个逻辑块又分割成了 N 份，也就是说如果一个逻辑块是 4KB(一般文件系统都使用这个值)，在有 5 块盘的 RAID 3 系统中其中 4 块是数据盘，1 块是校验盘。这样，把 4KB 分成 4 块，每块 1KB，每个数据盘上各占 1 块，也就是两个扇区。而文件系统下发的一个 IO，至少是以一个逻辑块为单位的，也就是不能 IO 半个逻辑块的单位，也就不可能存在一个 IO，大小是小于 4KB 的，要么是 1 个 4KB，要么是 N 个 4KB。但这只是针对文件系统下发的 IO，磁盘控制器驱动向磁盘下发的 IO 最细粒度可以为一个扇区。这样，就保证了文件系统下发的一次 IO，不管多大都被跨越了所有数据盘。

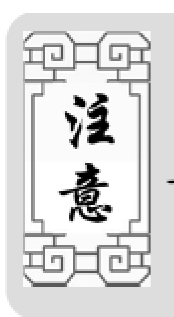
读又分成连续读和随机读。连续读指的是每个 IO 所需要提取的数据块在序号上是连续的，磁头不必耗费太多时间来回寻道，所以这种情况下寻道消耗的时间就很短。我们知道，一个 IO 所用的时间约等于寻道时间加上数据传输时间。IOPS=1/(寻道时间+数据传输时间)，由于寻道时间相对于传输时间要大几个数量级，所以影响 IOPS 的关键因素就是寻道时间。而在连续 IO 的情况下，仅在换磁道时候需要寻道，而磁道都是相邻的，所以寻道时间也足够短。在这个前提下，传输时间这个分母就显示出作用来了。由于 RAID 3 是一个 IO，必定平均分摊到了 N 个数据盘上，所以数据传输时间是单盘的 1/N，从而在连续 IO 的情况下，大大增加了 IOPS。而磁盘总体传输速率约等于 IOPS 乘以 IO SIZE。不管 IO SIZE 多大，RAID 3 持续读性能几乎就是单盘的 N 倍，非常强大。

再看看持续写，同样的道理，写 IO 也必定分摊到所有数据盘，那么寻道时间也足够短(因为是持续 IO)。所以写的时候所耗费的时间也是单盘的 1/N，因此速率也是单盘的 N 倍。有人说 RAID 3 的校验盘是热点盘，是瓶颈。理由是 RAID 3 写校验的时候，需要像 RAID 5 一样，先读出来原来的校验块，再读出原来的数据块，接着计算出新校验，然后写入新数据和新校验。实际上 RAID 3 中每个 IO 必定要改动所有数据盘的数据分块。因为一个文件系统 IO 的块已经被分割到所有盘了，只要这个 IO 是写的动作，那么物理磁盘上的所有分块，就必定要全部都被更新重写。既然这样，还有“旧数据”和“旧校验”的概念么？没有了，因为这个 IO 上的所有分割块需要全部被更新，包括校验块。数据在一次写入之前，控制器就会计算好校验块，然后同时将数据块和校验块写入磁盘。这就没有了什么瓶颈和热点的区别？

RAID 4 是有热点盘，因为 RAID 4 系统处理文件系统 IO 不是每次都会更新所有盘的，所以它必须用 RAID 5 的那个计算新校验的公式，也就是多出 4 个操作那个步骤，所以当然有瓶颈了！要说 RAID 3 有热点盘，也行，所有盘都是热点盘，数据、校验，所有盘，对 RAID 3 来说，每次 IO 必将牵动所有盘，那么就可以说 RAID 3 全部都是热点盘！

再来看看 RAID 3 的随机读写。所谓随机 IO，即每次 IO 的数据块是分布在磁盘的各个位置，这些位置是不连续的，或者连续几率很小。这样，磁头就必须不断地换道，换道操作是磁盘操作中最慢的环节。根据公式 IOPS=1/(换道时间+数据传输时间)，随机 IO 的时候换道时间很大，大出传输时间几个数量级，所以传输得再快，翻 10 倍也才增高了一个数量级，远不及换道时间的影响大，所以此时可以忽略传输时间的增加效应。由于一次 IO 同样是被分割到了所有数据盘，那么多块盘同时换道，然后同时传输各自的那个分块，换道时间就约等于单盘。传输时间是单盘的 1/N，而传输时间带来的增效可以忽略。所以对于随机读写的性能，RAID 3 并没有提升，和单盘一样，甚至不及单盘。因为有时候磁盘不是严格主轴同步的，这样换道慢的磁盘会拖累其他磁盘。

再来看看并发 IO。显然，RAID 3 执行一次 IO 必将牵动占用所有盘，那么此时其他排队的 IO 就必须等待，所以 RAID 3 根本就不能并发 IO。



上文中的“IO”均指文件系统下发的 IO，而不是指最终的磁盘 IO。

4.1.5 RAID 4 阵式

七星自从学习了西方先进的基础科学之后，一发而不可收！以前已经是以钻研为乐，现在成了以钻研为生了。以前饿了还知道去要饭吃，现在七星已经感觉不到饿了，只要有东西让他钻研，就等于吃饭了。

又话说某天七星正在闭目思考修炼，他回想起了双刀时代的辉煌，独行时代的凄苦和优雅剑时代的中庸之乐。往事历历在目，再看看如今已经是穷困潦倒的自己，他不禁潸然泪下，老泪纵横。

他给上面的三种思想，分别划分了门派，RAID 0 属于激进派，RAID 1 属于保守派，RAID 2 和 RAID 3 属于中庸派。中庸派的思想一方面汲取了激进派的横分竖割提高威力的做法，一方面适当地降低威力来向保守派汲取了自保的经验，而创立了引以为豪的校验盘的绝妙技术。

七星想着激进派似乎已经没有什么可以让中庸派值得借鉴的地方了，倒是保守派的一个关键技术中庸派还没有移植过来，那就是同时应付多个敌人的技术。虽然当年独行侠根本就没有意识到他的独孤影子掌可以同时应付两个敌人，因为独行侠一生都没有同时和两个人交过手。虽然独孤掌的秘笈中也没有提及这门绝招，但是七星凭他积累多年的知识和经验，强烈地感觉到并发 IO 早在独孤掌时代就已经被实现了，只不过没有被记载，而且一直被人忽略！要想有所突破就必须突破这一关！想到这里，七星立即再次开始了他的实验。

RAID 2 阵式中，数据块被以位为单位打散在多块磁盘上存储，这种设计确实应该被淘汰了，且不说IO设计合理与否，看它校验盘的数量就让人气不打一处来。那么再看看 RAID 3，在 RAID 3 的 IO 设计中还是走了 RAID 2 的老路子，也就是一次 IO 尽量让每块磁盘都参与，而控制器的一次 IO 数据块不会很大，那么想让每块磁盘都参与这个 IO，就只能人为地减小条带深度的大小。

事实证明这种 IO 设计在 IOSIZE/IOPS(比值)很大的时候，确实效果明显。但是现实应用中，很多应用的 IOSIZE/IOPS 都很小，比如随机小块读写等，这种应用每秒产生的 IO 数目很大，但是每个 IO 所请求的数据长度却很短。如果所有磁盘同一时刻都被一个 IO 占用着，且不能并发 IO，只能一个 IO 一个 IO 的来做。由于 IO 块长度小，此时全盘联动来传输这个 IO，得不偿失，还不如让这个 IO 的数据直接写入一块磁盘，空余的磁盘就可以做其他 IO 了。

要实现并发 IO，就需要保证有空闲的磁盘未被 IO 占用，以便其他 IO 去占有磁盘进行访问。惟一可以实现这个目的的方法就是增大条带深度，控制器的一个 IO 过来，如果这个 IO 块小于条带深度，那么这次 IO 就被完全“禁锢”在一块磁盘上，直接就写入了一个磁盘上的 Segment 中，这个过程只用到了一块磁盘。而其他 IO 也可以和这个 IO 同时进行，前提是其他 IO 的目标不是这个 IO 要写入或读取的磁盘。所以实现 IO 并发还需要增大数据的随机分布性，而不要连续在一个磁盘上分布。这里七星大侠忽略了一个非常重要的地方，下面会看到。

在这些分析的基础上，七星将 RAID 3 进行了简单的改造，增大了条带深度，于是便创立了一个新的阵式，名曰 RAID 4。

四海一家

并肩携手并存并取海誓山盟！

先来后到先进先出天经地义！

4.1.6 RAID 5 阵式

话说七星大侠正在为创立了 RAID 4 阵式而欢喜的时候，麻烦来了。很多江湖上的朋友都给他捎信说，修炼了 RAID 4 阵式之后，在 IO 写的时候，好像性能相对于 RAID 3 并没有什么提升，不管 IOSIZE/IOPS 的值是多少。这个奇怪的问题，让七星大侠天天如坐针毡、茶饭不思，终日思考这个问题的原因。他不停地拿着两块磁盘和一张纸(校验盘)比划。时间一长，七星有一天突然发现，纸已经被他画的不成样子了，需要另换一张。这引起了思维活跃的七星大侠的思考，“并发 IO，并发 IO，并发 IO，……”，他不停地在嘴里念叨着，突然他两眼一睁，骂了一句之后，开始奋笔疾书。

七星大侠想到了什么让他恍然大悟呢？原来，七星经过思考之后，发现 RAID 4 确实是他的一大败笔，相对 RAID 3 没有什么性能提升，反而误人子弟，浪费了很多人的时间去修炼一个无用的功夫。

七星创立 RAID 4 时，太过大意了，竟然忽略了一件事情。每个 IO 写操作必须占用校验盘，校验盘每一时刻总是被一个 IO 占用，因为写数据盘的时候，同时也要读写校验盘上的校验码。所以每个写 IO 不管占用了哪块数据盘，校验盘它是必须占用的，这样校验盘就成为了瓶颈，而且每个写入 IO 都会拖累校验盘，使得校验盘没有休息的时间，成了“热点盘”，非常容易损坏。

没有引入校验功能，数据盘可以被写 IO 并发，引入了校验功能之后，数据盘还可以并发，但是校验盘不可以并发，所以整体上还是不能并发。除非不使用校验盘，不过那就和 RAID 0 没什么区别了。所以七星在 RAID 4 上掉进了一个误区，如今他终于醒悟了。

创立的 RAID 4 什么都不是，不伦不类。七星这个郁闷啊，为了实现真正的写 IO 并发，他这次是豁出去了，一定要创立新的阵式！

思考 RAID 4 的关键错误在于忽略了校验盘。每个 IO 不管目标在哪个数据盘，但是一定要读写校验盘。而校验盘只有一块，不读也得读！那如果有两块校验盘，它能否随机选择一块来读写？不行，这两块校验盘之间也要同步起来，类似 RAID 1，这样开销太大，成本太高。

不妨作一下演绎，首先，我们的目标是并发 IO。要并发 IO，校验盘某一时刻必须可以被多个 IO 占用，这是必须的，否则就不是并发 IO。但是“校验盘某一时刻可以被多个 IO 占用”，这不简直是扯淡么？一块磁盘怎么可以同时被多个 IO 占用呢？所以七星下了结论，中庸派不可能实现并发 IO。

结论下了，七星也病了，彻彻底底的病倒了。他不甘心，在他心中一定有一个完美的阵式。他拿着那张已经快被画烂的纸，气愤至极，将纸撕成了两半。碎片就像七星那破碎的心，飘飘落下，不偏不倚，正好分别落到了地上的两把剑上，分别盖住了剑的一半。七星看着这情景，一发愣，仿佛冥冥中一直有个神仙在指引着他似的，又一次让七星茅塞顿开，恍然大悟。“老天助我啊！哇哈哈哈哈哈！”

七星疯了一般从炕上滚落下来，他又找来一把剑，把纸撕成三块，分别盖住每把剑的三分之一。同样，4 把剑，把纸撕成 4 块，盖住剑的四分之一。良久之后，七星仰天长叹：“完美，太完美了！”。

七星赶紧静下心来，他深知，必须经过深思熟虑，才不会出现问题，不能重蹈 RAID 4 的覆辙。他花了半个月的时间，用树枝在地上画图演算，并仔细分析。一块磁盘同一时刻不能被多个 IO 占用，这是绝对真理，不可质疑的真理。那么以前也曾经想过，把校验盘做成多块，可否？也不好，不完美。这次把校验盘分割开，组合于数据盘之中，依附于数据盘，这样就完美的避开了那个真理。既然多个 IO 可以同时刻访问多块数据盘，而校验盘又被打散在各个数据盘上，那么就意味着多 IO 可以同时访问校验盘(的“残体”)。这样就大大增加了多 IO 并发的几率，纵使发生多个 IO 所要用到的校验盘的“残体”可能在同一块数据盘上，这样还是可以 IO 排队等待的。

如果数据盘足够多，校验盘打散的部分就会分布得足够广泛，多 IO 并发的几率就会显著增大！他根据这个推断做实验，首先是两个数据盘。把纸撕成两半，分别盖住两把剑的一半，这样实际的数据盘容量其实是一把剑的容量，校验盘容量也是一把剑的容量，它们分别占了总容量的二分之一。由于 2 块盘的 RAID 5 系统，对于写操作来说不能并发 IO，因为一个 IO 访问其中一块盘的数据的时候，校验信息必定在另一块盘，必定也要同时访问另一块盘。同样，3 块盘的 RAID 5 系统也不能并发 IO，最低可以并发 IO 的 RAID 5 系统需要 4 块盘，而此时最多可以并发两个 IO，可以算出并发几率为 0.0322。更多磁盘数量的 RAID 5 系统的并发几率将更高。图 4.6 为一个 RAID 5 系统的示意图。

七星这次可谓是红星高照，脸色红润，病态全无。他把这个新创立的阵式叫做 RAID 5。正可谓：

心似剑，剑如心，
剑心合，方不侵。
分久必合合久分，
分分合合天地真！

图4.6　RAID 5 系统示意图

RAID 5 也不是那么完美无缺的，可以说 RAID 5 是继 RAID 0 和 RAID 1 之后，第一个能实现并发 IO 的阵式，但是比 RAID 1 更加划算，比 RAID 0 更加安全。RAID 5 浪费的资源，在 2 块盘的系统中与 RAID 1 是一样的，都是二分之一。但是随着磁盘数量的增加，RAID 5 浪费的容量比例越来越小，为 N 分之一，而 RAID 1 则永远是二分之一。

RAID 5 和 RAID 0 都是利用条带来提升性能，但是它又克服了 RAID 0 的鲁莽急躁，对数据用校验的方式进行保护。但是 RAID 5 的设计思想，注定了它的连续读性能不如 RAID 3，RAID 3 由于条带深度很小，每次 IO 总是能牵动所有磁盘为它服务，对于大块连续数据的读写速度很快。但是 RAID 5 的条带深度比较大，每次 IO 一般只使用一块数据盘，而且通用 RAID 5 系统一般被设计为数据块都是先放满一个 Segment，再去下一个磁盘的 Segment 存放，块编号是横向进行的。

RAID 5 在随机读方面，确实是首屈一指的，这要归功于它的多 IO 并发的实现，这里指的是随机 IO。也就是说 RAID 3 在 IOSIZE 值大的时候具有高性能，RAID 5 在随机 IOPS 大的时候具有高性能。

RAID 5 的一大缺点就是写性能较差。写性能差是中庸派的通病，其根本原因在于它们每写一扇区的数据就要产生其校验扇区，一并写入校验盘。尤其是更改数据的时候，这种效应的影响尤其严重。

RAID 5 写的基本过程是这样的，新数据过来之后，控制器立即读取待更新扇区的原数据，同时也要读取这个条带上的校验数据。三者按照公式运算，便可得出新数据的校验数据，然后将新数据和新数据的校验数据写到磁盘。公式如下。

新数据的校验数据=(老数据 EOR 新数据) EOR 老校验数据

鱼和熊掌不可兼得！最后七星总结了这么一句话。也就是说随机并发 IO 和写性能二者取其一。但是有些文件系统巧妙的减少了这种写惩罚，使得 RAID 4 的缺点被成功削减，从而将其优点显现了出来，本章后面会提到这种优化操作。



RAID 5 一次写的动作，其实要浪费掉 3 个其他动作，也就是要先读出老数据，读出老校验数据，然后写新数据和校验数据，这样只有“写新数据”是要完成的目的，而捎带了三个额外操作。纵观 RAID 0 和 RAID 1 此二者，RAID 0 鲁莽，写就是写，不带任何考虑，所以速度最快。RAID 1 自保，但是每次也只要写两次即可，只是额外多了一个操作。所以 RAID 5 和 RAID 4 在处理写方面是失败的。就连 RAID 2 和 RAID 3 都比 RAID 5 写性能强，因为它们的条带深度很小，任何一次正常点的 IO 几乎会覆写所有盘，均会将这整个条带上的位都改变。所以 RAID 2 和 RAID 3 不用顾忌条带上是否还有未被更新的数据，所以它不会管老数据如何，只管从新数据计算出新校验数据，然后同时将数据位和校验位分别写到数据盘和校验盘，这样只用了 2 个操作，比 RAID 5 少了两次读的过程。

RAID 5E 和 RAID 5EE 阵式

七星大侠推出 RAID 5 之后，受到了极为广泛地应用，江湖上的武林人士都在修炼，有些练成的大侠还各自创办了数据库、网站等生意，得益于 RAID 5 的随机 IO 并发特性，这些人赚了一大笔，生意火的一塌糊涂。然而，七星还是那个要饭的七星，剑还是那把剑，依然终日以钻研为乐，以钻研为生。

话说有一天，有个侠客专门找到了七星大侠，侠客请他到“纵横斋”煮酒畅饮。酒过三巡，菜过五味，侠客进入了正题，向七星叙述了一件事情。他说他已经炼成了 RAID 5 阵式，但是在使用的时候总是心里没底。其原因就是一旦一块磁盘损坏，虽然此时不影响使用，但是总有顾虑，不敢全力出招，就怕此时再坏一块磁盘，整个阵式就崩溃了。

他请求七星决这个问题，临走的时候留下了几块市面上品质最好的硬盘和一些银子供七星研究使用。七星很是感动，几十年来从来没有一位江湖人士和他交流切磋过，也从来没有一个人来帮助过他。此景让他泪流满面，感动得不知说什么好。他向那位侠客道：“能交您这位豪杰人士，我七星此生无憾！”。随后，七星又开始终日研究。这位大侠就是几十年后谱写降龙传说的张真人。



嗯，一旦一块磁盘损坏，此时这块盘上的数据已经不复存在，但是如果有 IO 请求这块坏盘上的数据，那么可以用还存在的数据，校验出这块损坏的数据，传送出去。也就是说，损坏的数据是边校验边传送，现生成现传送的。而对于要写入这块盘的 IO，控制器会经过计算，将其“重定向”到其他盘。所谓重定向并不是完全透明的写入其他盘，而是运用 XOR 进行逆运算，将写入的数据代入算式进行逆算，得到的结果写入现存的磁盘上。

此时如果再有一块磁盘损坏，无疑阵式就要崩溃了。解决这个问题的直接办法，就是找一块备用的磁盘。一旦有磁盘损坏，其他磁盘立即校验出损坏的数据，立即写到备用磁盘上。写完之后，阵形就恢复原样了，就没有顾虑了。但是必须保证在其他磁盘齐力校验恢复数据的过程中，不可再有第二块磁盘损坏，不然便会玉石俱焚！

想到这里，七星开始了他的实验，并且取得成功。就是在整个阵式中增加一块热备盘，平时这块磁盘并不参与组阵，只是在旁边观战，什么也不干。一旦阵中某个人受伤不能参战了，这个热备盘立即顶替它，其他人再把功力传授给它。传授完毕后，就像原来的阵式一样了。如果在大家传授功力的时候，有 IO 请求这块损坏磁盘上的数据，那么大家就暂停传授，先应付外来的敌人。当没有来针对这块损坏磁盘的挑衅时，大家会继续传授。

七星的经验不断丰富，他知道不能急躁，所以实验成功之后，七星并没有马上通知那位大侠，而是继续在想有没有可以改进的地方。他想，热备盘平时不参与组阵，那就不能称作阵式的一部分，而是被排斥在外。这块磁盘平时也没有 IO，起不到作用，这样就等于浪费了一块磁盘。那么是不是考虑也让它参与到阵形中来呢？如果要参与进来，那让它担任什么角色呢？热备角色？如果没有人受伤，这个角色在阵中只会是个累赘。怎么办好呢？七星忽然掠过一丝想法，是否可以让阵中各个角色担待一下，从各自的领地保留出一块空间，用作热备盘的角色呢？

这样把热备盘分布在各个磁盘上，就不会形成累追，并且同时解决了热备盘和大家不协调的问题。说干就干，七星给那位大侠写了一封信，信中称这种阵式为 RAID 5E。七星继续琢磨 RAID 5E，让阵中每个人都保留一块领地，而不横割，虽然可以做到数据的及时备份，但是这块领地总显得不伦不类。七星突然想到被撕碎的纸片飘然落下的情景，忽然计上心头！既然校验盘都可以横分竖割的融合到数据盘，为什么热备盘不能呢？一样可以！七星想到这里，于是又给那位侠客去了一封信，信中描述这种新的阵式为 RAID 5EE。

那位侠客给七星回了一封信：

七星转，北斗移，
英雄无谓千万里。
待到再次相见时，
白发苍，叙知己！

七星看后老泪纵横，颇为感动。相见恨晚啊，到了晚年才遇到人生知己！

4.1.7 RAID 6 阵式

如今，七星已经从一个壮年小伙变成了个孤苦伶仃的老头。回想他的一生，从 RAID 0 一直到 RAID 5(E、EE)创立了 6 种阵式，各种阵式各有所长。他最得意的恐怕就是中庸派的，中庸之道。可就是这个中庸之道，却还有一个一直也未能解决的问题，那就是其中任何一种阵式，都最多同时允许损坏一块磁盘，如果同时损坏多块，整个阵不攻自破！七星想到这里就一阵酸楚。已经是白发苍苍的七星老侠，决定要用晚年最后一点精力来攻破这个难题。

七星老侠一生精研阵式，有很多宝贵的经验。他这次采用了逆向思维，假设这个模型已经做好，然后从逆向分析他是怎么作用的。先描绘出多种模型，然后一个一个地去攻破，找出最适合模型。

七星描绘了这么一个模型，假设有 5 块盘组成一个 RAID 阵列，4 块数据盘，一块校验盘，那么同一时刻突然 4 块数据盘中的两块损坏作废，只剩下两块数据盘和一块校验盘。假设有某种算法，可以恢复这两块丢失磁盘上的数据，那么怎么从这个模型推断出，这个

算法是怎么把丢失的两块盘数据都恢复出来的呢？七星冥思苦想。七星在幼年学习方程的时候，知道要求解一个未知数，只需知道只包含这一个未知数的一个等式即可逆向求解。就像布尔的逻辑算式用在 RAID 3 阵式时候一样，各个数据盘上的数据互相 XOR 之后就等于校验盘上的校验数据，这就是一个等式。

```
D1 XOR D2 XOR D3 = Parity
```

如果此时 D1 未知，而其他 3 个值都已知，那么就可以逆向解出未知数，而这也是 RAID 3 进行校验恢复的时候所做的。那么此时如果 D1 和 D2 都未知，也就是 1 号盘和 2 号盘都损坏了，还能解出这两个值么？数学告诉七星，这是不可能的，除非除了这个等式还额外存在一个和这个等式不相关的另一个等式！

要求解两个未知数，只要知道关于这两个未知数的不相关的两个关系方程即可。比如有等式为 D1 ？ D2 ？ D3 = *，联立以上两个等式，即可求出 D1 和 D2。七星开始寻觅这个等式，这个等式是已经存在？还是需要自己去发明呢？七星一开始打算从布尔等式找寻出第二个等式的蛛丝马迹，但是后来他根据因果率知道如果从布尔等式推出其他某些等式，那么推出的等式和布尔等式就是相关的。互相相关的两个等式，在数学上是等价的，无法作为第二个等式。七星有所察觉了，他认为要想得出第二个等式，必须由自己发明一套算法，一套和布尔等式不相关的算法！他开始在纸上演义算法，首先他开始从算术的加减方程开始着手，他写出了一个可以求解两个未知数的二元方程。

```
X+Y=10
2X+3Y=20
```

这算是最简单的算术方程了。可以求得 x=10，y=0。

以上是对于算术运算的方程，那么布尔逻辑运算是否也可以有方程呢？七星写下如下的式子。

```
x XOR y = 1
Ax XOR By = 0
```

第一个方程已经存在了，也就是用在 RAID 3 上的校验方程。第二个方程是七星模仿加减方程来写的，也就是给 x 和 y 两个值分别加上了一个系数，而这两个系数不能是从第一个等式推得的，比如是将第一个等式未知数的系数同乘或同除以某个数得出来的，这样就是相关等式了。七星立即找来布尔逻辑运算方面的书深入学习，终于得到了印证，这种方程确实存在！七星激动得跳了起来！他立即投入到研发中。过了两个月，终于得出了结果！大获成功！

七星对一份数据使用两套算法各自算出一个等式，1 号等式右边的结果写入校验盘 1，2 号等式右边的值写入校验盘 2。这样，只要使用中有两个值发生丢失，就可以通过这两个等式联立，解出丢失的两个值。不管这两个值是等式左边的还是等式右边的，只要代入这两个等式中，就可以求出解。

数学的力量是伟大的，任何东西只要通过了数学的验证，就是永恒的！

同样，七星将用在 RAID 5 中的方法，用在了新创立的阵式中，将校验盘分布到数据盘中，不同的是新阵式的校验盘有两块，在每块磁盘上放置两个等式需要的校验值。

七星给这种阵式取名为 RAID 6。

RAID 6 相对其他各种中庸派的阵式安全多了，但同时它的写性能更差了，因为它要多读出一个校验数据，而且计算后还要写入一次，这就比 RAID 5 每次写耗费多了 2 个操作，变成了 6 次操作，所谓的“写惩罚”更大了。确实是鱼与熊掌不能兼得啊！正可谓：

寻寻觅觅终冷清，
七星北斗伴我行。
世间万物皆规律，
求得心法谋太平！

七星在创立了 RAID 6 阵式之后，已经老态龙钟，疾病缠身，所剩时日不多。而江湖上可是一派热闹，修炼的修炼，修炼好的就立门派，开设数据库、网站等服务来大赚钱财。而打着七星大侠旗号到处招摇撞骗的人也不在少数。张真人路见不平拔刀相助，他召开武林大会，表彰了七星的功绩，指出真正的七星现在早已归隐，那些招摇撞骗的人也就没有容身之地了。他还提议将七星所创立的 7 种阵式命名为“七星北斗阵”，以纪念他和七星之间的北斗豪情！

张真人亲自到深山去探望已经是下不了床的七星，并将武林中发生的事情告诉了他。此后没几天，七星无憾地离开了人世，化作了北斗七星，在天上洒下无限的光芒照耀世间！张真人给老人办了后事，并将七星过世的消息宣布了出去。没想到第二天，七星、星七、北斗、斗北、七星北斗、北斗七星、星七北斗、星七斗北等商标就被抢注了！大批的商人在发着横财。正可谓：

七星赞

七星阵里论七星，
北斗光前参北斗。
不知天上七星侠，
如今过活要饭否？

4.2 七种 RAID 技术详解

下面从纯技术角度，深入剖析目前存在的七种 RAID 模式的组成原理和结构，并分析各种级别相对于单盘 IO 速率的变化。

首先澄清一点，所谓 Stripe 完全是由程序在内存中虚拟出来的，说白了就是一个 map 公式。即仿佛是给程序戴了一个特殊的眼镜，程序戴上这个眼镜，就能看到“条”和“带”，就会知道将数据分布到条带上了。一旦摘下这个眼镜，那么看到的就是普通的物理磁盘扇区。这个眼镜就是实现 RAID 的程序代码。物理磁盘上根本不存在什么“条”和“带”，只有扇区。另外，程序会在磁盘特定的一些扇区中写入自己运行时需要的信息，比如一些 RAID 标签信息等。

图 4.7 所示为一个 RAID 0 系统的示意图。

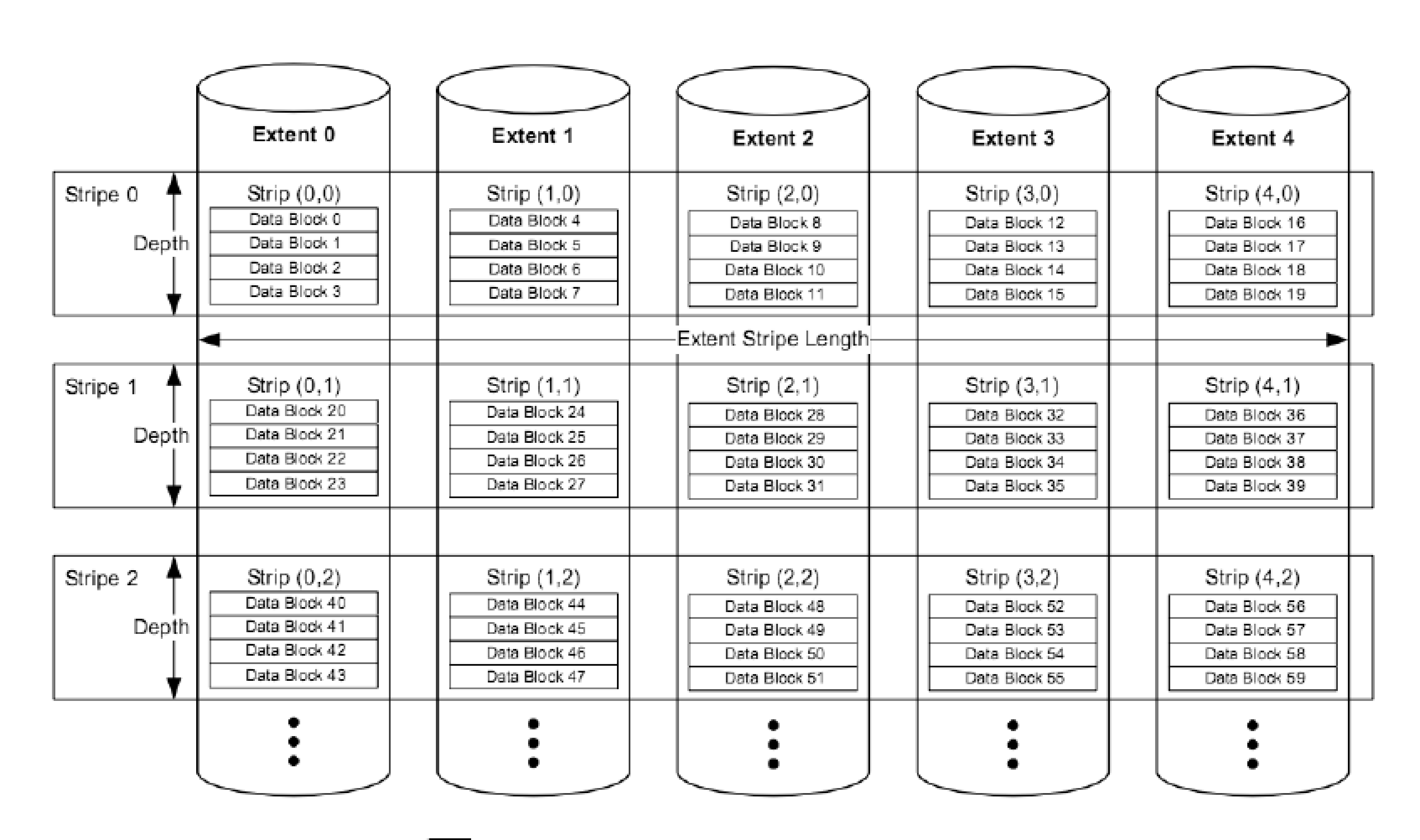


图4.7 一个典型的 RAID 0 系统

扇区、块、段(Segment)、条带、条带长度和深度

图 4.7 中的 5 个竖条，分别代表 5 个磁盘。然后在磁盘相同偏移处横向逻辑分割，形成 Stripee。一个 Stripee 横跨过的扇区或块的个数或字节容量，就是条带长度，即 Stripee length。而一个 Stripee 所占用的单块磁盘上的区域，称为一个 Segment。一个 Segment 中所包含的 data Block 或者扇区的个数或者字节容量，称为 Stripee depth。Data Block 可以是 N 倍个扇区大小的容量，应该可调，或者不可调，由控制器而定。

RAID 0 便是将一系列连续编号的 data Block 分布到多个物理磁盘上，扩散 IO 提高性能。其分布的方式，如图 4.7 所示。这个例子中，条带深度为 4，则 0、1、2、3 号 data Block 被放置到第一个条带的第一个 Segment 中，然后 4、5、6、7 号 Block 放置到第一个条带的第二个 Segment 中，依此类推，条带 1 放满后，继续放条带 2。这种特性称为“局部连续”，因为 Block 只有在一个 Segment 中是物理连续的，逻辑连续就需要跨物理磁盘了。

关于几个与 IO 相关的重要概念

IO 可以分为读/写 IO、大/小块 IO、连续/随机 IO、顺序/并发 IO。下面来分别介绍这几种 IO。

- 读/写 IO：这个就不用多说了，读 IO 就是发指令从磁盘读取某段序号连续的扇区的内容。指令一般是通知磁盘开始扇区位置，然后给出需要从这个初始扇区往后读取的连续扇区个数，同时给出动作是读还是写。磁盘收到这条指令就会按照指令的要求读或者写数据。控制器发出这种指令加数据并得到对方回执的过程就是一次 IO 读或 IO 写。注意，一个 IO 所要提取的扇区段一定是连续的，如果想提取或写入两段不连续的扇区段，只能将它们放入两个 IO 中分别执行，这也就是为何随机 IO 对设备的 IOPS 指标要求较高的原因。
- 大/小块 IO：指控制器的指令中给出的连续读取扇区数目的多少。如果数目很大，如 128、64 等，就应该算是大块 IO。如果很小，比如 1、4、8 等，就应该算是小块 IO。大块和小块之间，没有明确的界限。
- 连续/随机 IO：连续和随机是指本次 IO 给出的初始扇区地址和上一次 IO 的结束扇区地址是不是完全连续的或者相隔不多的。如果是，则本次 IO 应该算是一个连续

IO。如果相差太大，则算一次随机 IO。连续 IO 因为本次初始扇区和上次结束扇区相隔很近，则磁头几乎不用换道或换道时间极短。如果相差太大，则磁头需要很长的换道时间。如果随机 IO 很多，会导致磁头不停换道，效率大大降低。

- 顺序/并发 IO：意思是，磁盘控制器如果可以同时对一个 RAID 系统中的多块磁盘同时发送 IO 指令(当然这里的同时是宏观的概念，如果所有磁盘都在一个总线或者环路上，则这里的同时就是指向一块磁盘发送一条指令后不必等待它回应接着向另一块磁盘发送 IO 指令)，并且这些最底层的 IO 数据包含了文件系统级下发的多个 IO 的数据，则为并发 IO。如果这些直接发向磁盘的 IO 只包含了文件系统下发的一个 IO 的数据，则此时为顺序 IO，即控制器缓存中的文件系统下发的 IO 队列，只能一个一个来。并发 IO 模式在特定的条件下可以很大的提高效率和速度。
- IO 并发几率：单盘，IO 并发几率为 0，因为一块磁盘同时只可以进行一次 IO。对于 RAID 0，在 2 块盘情况下，条带深度比较大的时候(条带太小不能并发 IO)，并发 2 个 IO 的几率为 1/2。其他情况请自行运算。
- IOPS：完成一次 IO 所用的时间=寻道时间+数据传输时间，IOPS=IO 并发系数/(寻道时间+数据传输时间)。由于寻道时间相对于传输时间要大几个数量级，所以影响 IOPS 的关键因素就是降低寻道时间。在连续 IO 的情况下，寻道时间很短，仅在换磁道时候需要寻道。在这个前提下，传输时间越少，IOPS 就越高。
- 每秒 IO 吞吐量：显然，每秒 IO 吞吐量=IOPS × 平均 IO SIZE。IO SIZE 越大，IOPS 越高，每秒 IO 吞吐量就越高。设磁头每秒读写数据速度为 V，V 为定值。则 IOPS=IO 并发系数/(寻道时间+IO SIZE/V)。代入得每秒 IO 吞吐量=IO 并发系数 × IO SIZE × V/(V × 寻道时间+IO SIZE)。可以看出影响每秒 IO 吞吐量的最大因素就是 IO SIZE 和寻道时间。IO SIZE 越大，寻道时间越小，吞吐量越高。相比能显著影响 IOPS 的因素，只有一个就是寻道时间。

4.2.1 RAID 0 技术详析

RAID 0 是这样一种模式：我们拿 5 块盘的 RAID 0 为例子，如图 4.7 所示。

对外来说，参与形成 RAID 0 的各个物理盘会组成一个逻辑上连续，物理上也连续的虚拟磁盘。一级磁盘控制器(指使用这个虚拟磁盘的控制器，如果某台主机使用适配卡链接外部盘阵，则指的就是主机上的磁盘控制器)对这个虚拟磁盘发出的指令，都被 RAID 控制器收到并分析处理，根据 Block 映射关系算法公式转换成对组成 RAID 0 的各个物理盘的真实物理磁盘 IO 请求指令，收集或写入数据之后，再提交给主机磁盘控制器。

图 4.8 为一个 RAID 0 虚拟磁盘的示意图。

RAID 0 还有另一种非条带化模式，即写满其中一块物理磁盘之后，再接着写另一块，直到所有组成的磁盘全部写满。这种模式对 IO 写没有任何优化，但是对 IO 读能提高一定的并发 IO 读几率。

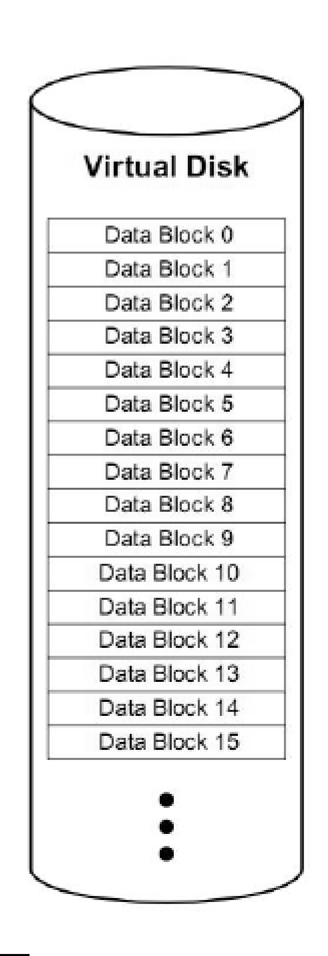


图 4.8 虚拟磁盘

访问 RAID 0 磁盘

下面我们来具体分析一个从上到下访问 RAID 0 磁盘的过程。

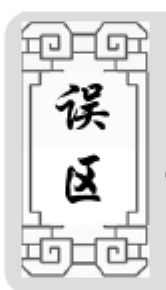
1】 **假如某一时刻，主机控制器发出指令：读取 初始扇区 10000 长度 128。**

2】 **RAID 控制器接受到这个指令之后，立即进行计算，根据对应公式(这个公式是 RAID 控制器在做逻辑条带化的时候制定的)算出 10000 号逻辑扇区所对应的物理磁盘的扇区号。**

3】 **依次计算出逻辑上连续的下 128 个扇区所在物理磁盘的扇区号。**

4】 **分别向对应这些扇区的磁盘，再次发出指令。这次是真实的读取数据了，磁盘接受到指令，各自将数据提交给 RAID 控制器，经过控制器在 Cache 中的组合，再提交给主机控制器。**

分析以上过程，发现如果这 128 个扇区都落在同一个 Segment 中的话，也就是说条带深度容量大于 128 个扇区的容量(64KB)，则这次 IO 就只能真实的从这一块物理盘上读取，性能和单盘相比会减慢，因为没有任何优化，反而还增加了 RAID 控制器额外的计算开销。所以，在某种特定条件下要提升性能，让一个 IO 尽量扩散到多块物理盘上，就要减小条带深度。在磁盘数量不变的条件下，也就是减小条带大小 Stripe SIZE(也就是条带长度)。让这个 IO 的数据被控制器分割，同时放满一个条带的第一个 Segment、第二个 Segment 等，依此类推，这样就能极大的占用多块物理盘。



总是以为控制器是先放满第一个 Segment，再放满第二个 Segment。

其实是同时进行的，因为控制器把每块盘要写入或者读取的数据都计算好了。如果这些目标磁盘不在相同的总线中，那么这种宏观“同时”的力度将会更加细。因为毕竟计算机总线是共享的，一个时刻只能对一个外设进行 IO。

所以，RAID 0 要提升性能，条带做的越小越好。但是又一个矛盾出现了，就是条带太小，导致并发 IO 几率降低。因为如果条带太小，则每次 IO 一定会占用大部分物理盘，队列中的 IO 就只能等待这次 IO 结束后才能使用物理盘。而条带太大，又不能充分提高传输速度，这两个是一对矛盾，要根据需求来采用不同的方式。如果随机小块 IO 多，则适当加大条带深度，如果连续大块 IO 多，则适当减小条带深度。

接着分析 RAID 0 相对于单盘的性能变化。根据以上总结出来的公式，可以推出表 4.1。

表 4.1 RAID 0 相对单盘的 IO 对比

RAID 0 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE/Stripe SIZE 较大	不支持	不支持	提升极小	提升了 N×系数倍	不支持	不支持	提升极小	提升了 N×系数倍

续表

RAID 0 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE/Stripe SIZE 较小	提升了(1+并发系数)倍	提升了(1+并发系数+系数)系数倍	提升极小	提升了系数倍	提升了(1+并发系数)倍	提升了(1+并发系数+系数)倍	提升极小	提升了系数倍

注：并发 IO 和 IO SIZE/Stripe SIZE 是一对矛盾，两者总是对立的。N=组成 RAID 0 的磁盘数目。系数=IO SIZE/Stripe SIZE 和初始 LBA 地址所处的 Stripe 偏移综合系数，大于等于 1。并发系数=并发 IO 的数量。

4.2.2 RAID 1 技术详析

RAID 1 是这样一种模式，拿两块盘的例子来进行说明，如图 4.9 所示。

RAID 1 和 RAID 0 不同，RAID 0 对数据没有任何保护措施，每个 Block 都没有备份或者校验保护措施。RAID 1 对虚拟逻辑盘上的每个物理 Block，都在物理盘上有一份镜像备份，也就是说数据有两份。对于 RAID 1 的写 IO，速度不但没有提升，而且有所下降，因为数据要同时向多块物理盘写，时间以最慢的那个为准，因为是同步的。而对于 RAID 1 的读 IO 请求，不但可以并发，而且就算顺序 IO 的时候，控制器也可以像 RAID 0 一样，从两块物理盘上同时读数据，提升速度。RAID 1 可以没有 Stripe 的概念，当然也可以有。同样我们总结出表 4.2。

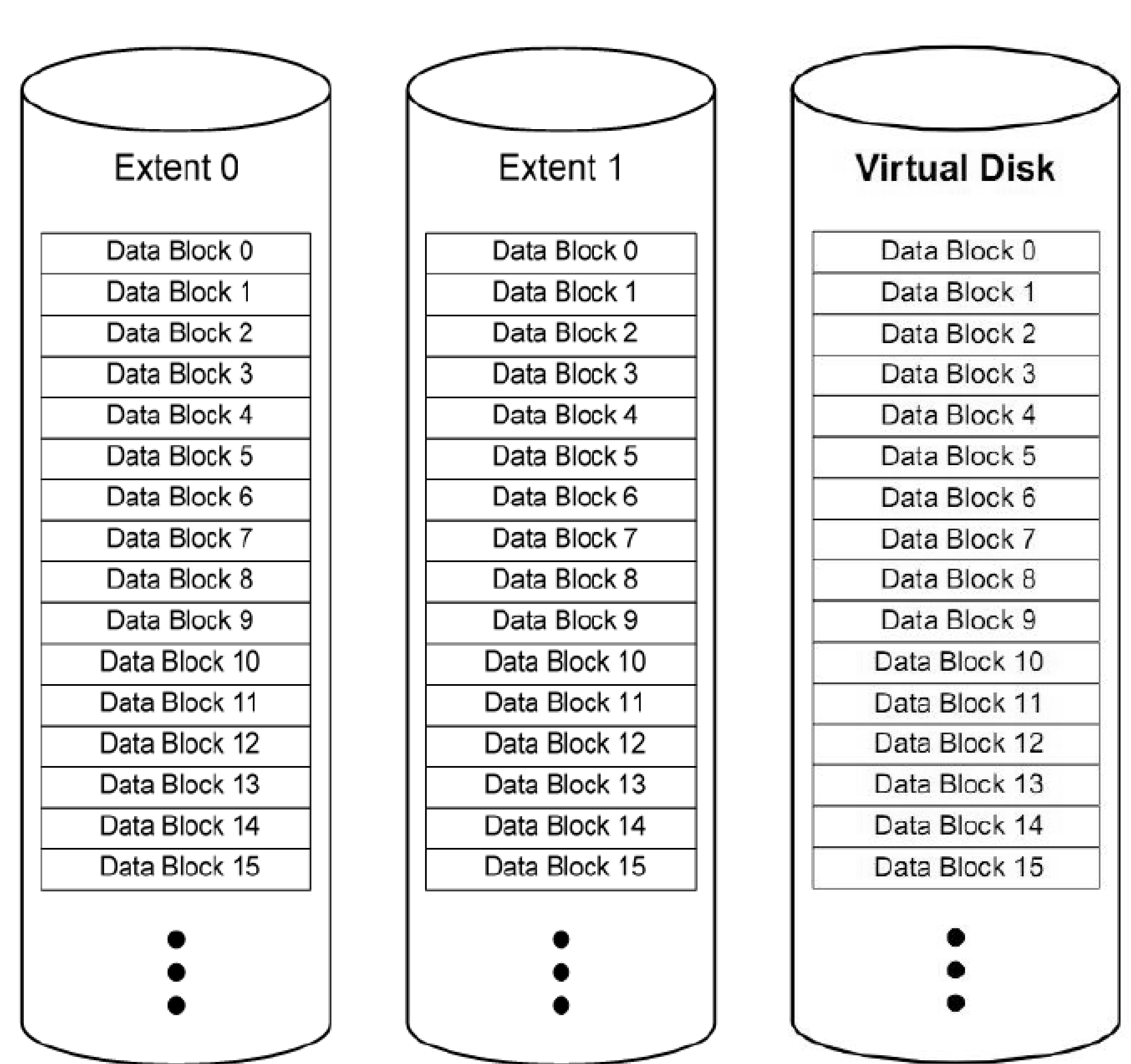


图4.9 RAID 1 系统示意图

表 4.2　RAID 1 系统相对于单盘的 IO 对比

RAID 1 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
	提升 N 倍或者并发系数倍	提升 N 倍或者并发系数倍	提升极小	提升了 N 倍	不支持	事务性 IO 可并发，提升并发系数倍	没有提升	没有提升

注：N=组成 RAID 1 镜像物理盘的数目。

在读、并发 IO 的模式下，由于可以并发 N 个 IO，每个 IO 占用一个物理盘，这就相当于提升了 N 倍的 IOPS。由于每个 IO 只独占了一个物理盘，所以数据传输速度相对于单盘并没有改变，所以不管是随机还是顺序 IO，相对单盘都不变。

在读、顺序 IO、随机 IO 模式下，由于 IO 不能并发，所以此时一个 IO 可以同时读取 N 个盘上的内容。但是在随机 IO 模式下，寻道时间影响很大，纵使同时分块读取多个磁盘的内容，也架不住寻道时间的抵消，所以性能提升极小。

在读、顺序 IO、连续 IO 模式下，寻道时间影响到了最低，此时传输速率为主要矛盾，同时读取多块磁盘的数据，时间减少为 1/N，所以性能提升了 N 倍。

写 IO 的时候和读 IO 情况相同，就不做分析了。写 IO 因为要同时向每块磁盘写入备份数据，所以不能并发 IO，也不能分块并行。但是如果控制器把优化算法做到极至的话，还是可以并发 IO 的，比如控制器从 IO 队列中提取连续的多个 IO，可以将这些 IO 合并，并发写入磁盘，前提这几个 IO 必须是事务性的，也就是说 LBA 必须连续，不然不能作为一个大的合并 IO。而且和文件系统也有关系，文件系统碎片越少，并发几率越高。

4.2.3　RAID 2 技术详析

RAID 2 是一种比较特殊的 RAID 模式，它是一种专用 RAID，现在早已被淘汰。它的基本思想是在 IO 到来之后，控制器将数据按照位分散开，顺序在每块磁盘中存取 1bit。这里有个疑问，磁盘的最小 IO 单位是扇区，有 512 字节，如何写入 1bit 呢？其实这个写入 1bit，并非只写入 1bit。我们知道上层 IO 可以先经过文件系统，然后才通过磁盘控制器驱动来向磁盘发出 IO。最终的 IO 大小，都是 N 倍的扇区，也就是 N×512 字节，N 大于等于 1，不可能发生 N 小于 1 的情况。即使需要的数据只有几个字节，那么也同样要读出或写入整个扇区，也就是 512 字节。

明白这个原则之后，再来看一下 RAID 2 中所谓的“每个磁盘写 1bit”是个什么概念。IO 最小单位为扇区(512 字节)，我们就拿一个 4 块数据盘和 3 块校验盘的 RAID 2 系统为例给大家说明一下。这个环境中，RAID 2 的一个条带大小是 4bit(1bit×4 块数据盘)，而 IO 最小单位是一个扇区，那么如果分别向每块盘写 1bit，就需要分别向每块盘写一个扇区，每个扇区只包含 1bit 有效数据，这显然是不可能的，因为太浪费空间，且没有意义。

我们拿以下 IO 请求为例。

1】 写入 初始扇区 10000　长度 1，这个 IO 目的是要向 LBA10000 写入一个扇区的数

据，也就是 512 字节。

2) RAID 2 控制器接受到这 512 字节的数据后，在 Cache 中计算需要写入的物理磁盘的信息，比如定位到物理扇区，分割数据成 bit。

3) 然后一次性写入物理磁盘扇区。也就是说第一块物理盘，控制器会写入本次 IO 数据的第 1、5、9、13、17、21 等位，第二块物理盘会写入 2、6、10、14、18、22 等位，其他两块物理盘同样方式写入。

直到这样将数据写完。我们可以计算出来，这 512 字节的数据写完之后，此时每块物理盘只包含 128 字节的数据，也就是一个扇区的四分之一，那么这个扇区剩余的部分，就是空的。

为了利用起这部分空间，等下次 IO 到来之后，控制器会对数据进行 bit 分割，将数据填入这些空白区域，控制器将首先读出原来的数据，然后和新数据合并之后，一并再写回这个扇区，这样做效率和速度都大打折扣。其实 RAID 2 就是将原本连续的一个扇区的数据，以位为单位，分割存放到不连续的多块物理盘上，因为这样可以在任意条件下都迫使其全磁盘组并行读写，提高性能，也就是说条带深度为 1 位。这种极端看上去有点做得过火了，这也是导致它最终被淘汰的原因之一。

RAID 2 系统中每个物理磁盘扇区其实是包含了 N 个扇区的"残体"。



那么如果出现需要更新这个 4 个扇区中某一个扇区的情况，怎么办？

这种情况下，必须先读出原来的数据，和新数据合并，然后在一并写入。其实这种情况出现的非常少。我们知道上层 IO 的产生，一般是需要先经过 OS 的文件系统，然后才到磁盘控制器这一层的。所以磁盘控制器产生的 IO 一般都是事务性的，也就是这个 IO 中的所有扇区很大几率上对于上层文件系统来说是一个完整的事务，所以很少会发生只针对这个事务中某一个原子进行读写的情况。

这样的话，每次 IO 就有很大几率都会包含入这些逻辑上连续的扇区的，所以不必担心经常会发生那种情况。即便发生了，控制器也只能按照那种低效率的做法来做，不过总体影响较小。但是如果随机 IO 比较多，那么这些 IO 初始 LBA，很有可能就会命中在一个两个事务交接的扇区处。这种情况就会导致速度和效率大大降低。连续 IO 出现这种情况的几率非常小了。

RAID 2 因为每次读写都需要全组磁盘联动，所以为了最大化其性能，最好保证每块磁盘主轴同步，使同一时刻每块磁盘磁头所处的扇区逻辑编号都一致，并存并取，达到最佳性能。如果不能同步，则会产生等待，影响速度。

基于 RAID 2 并存并取的特点，RAID 2 不能实现并发 IO，因为每次 IO 都占用了每块物理磁盘。

RAID 2 的校验盘对系统不产生瓶颈，但是会产生延迟，因为多了计算校验的动作。校验位和数据位是一同并行写入或者读取的。RAID 2 采用汉明码来校验数据，这种码可以判断修复一位错误的数据，并且使用校验盘的数量太多，4 块数据盘需要 3 块校验盘。但是随着数据盘数量的增多，校验盘所占的比例会显著减小。

RAID 2 和 RAID 0 有些不同，RAID 0 不能保证每次 IO 都是多磁盘并行，因为 RAID 0 的条带深度相对于 RAID 2 以位为单位来说是太大了。而 RAID 2 由于每次 IO 都保证是多磁盘并行，所以其数据传输率是单盘的 N 倍。为了最好的利用这个特性，就需要将这个特性的主导地位体现出来。

而根据 **IOPS=IO 并发系数/(寻道时间+数据传输时间)**，寻道时间比数据传输时间要大几个数量级。所以为了体现数据传输时间减少这个优点，就必须避免寻道时间的影响，而最佳做法就是尽量产生连续 IO 而不是随机 IO。所以，RAID 2 最适合连续 IO 的情况。另外，根据**每秒 IO 吞吐量=IO 并发系数×IO SIZE×V/(V×寻道时间+IO SIZE)**，如果将 IO SIZE 也增大，则每秒 IO 吞吐量也将显著提高。所以，RAID 2 最适合的应用就是产生连续 IO、大块 IO 的情况。不言而喻，视频流服务等应用适合 RAID 2。不过，RAID 2 的缺点太多，比如校验盘数量多、算法复杂等，它逐渐被 RAID 3 替代了。表 4.3 比较了 RAID 2 系统与单盘的性能。

表 4.3　RAID 2 系统相对单盘的 IO 对比

RAID 2 IOPS	读			写		
	顺序 IO			顺序 IO		
	非事务性随机 IO	事务性随机 IO	连续 IO	非事务性随机 IO	事务性随机 IO	连续 IO
IO 满足公式条件	提升极小	提升极小	提升 N 倍	性能降低	提升极小	提升 N 倍

注：N=数据盘数量。RAID 2 不能并发 IO。

4.2.4 RAID 3 技术详析

图 4.10 所示为一个 RAID 3 系统的条带布局图。

图 4.10　RAID 3 系统示意图

由于 RAID 2 缺点比较多，比如非事务性 IO 对它的影响、校验盘数量太多等。RAID 2 的劣势就在于它将数据以 bit 为单位进行分割，将原本物理连续的扇区转变成物理不连续，而逻辑连续的。这样就导致了它对非事务性 IO 的效率低下。为了从根本上解决这个问题，RAID 3 出现了。

既然要从根本上解决这个问题，首先就是需要抛弃 RAID 2 对扇区进行分散的做法，RAID 3 保留了扇区的物理连续。RAID 2 将数据以 bit 为单位分割，这样是为了保证每次 IO 占用全部磁盘的并行性。而 RAID 3 同样也保留了这个特点，但是没有以 bit 为单位来分散数据，而是以一个扇区或者几个扇区为单位来分散数据。RAID 3 还采用了高效的 XOR 校验

算法，但是这种算法只能判断数据是否有误，不能判断出哪一位有误，更不能修正错误。XOR 校验使得 RAID 3 可以不管多少块数据盘，只需要一块校验盘就足够了。

RAID 3 的每一个条带，其长度被设计为一个文件系统块的大小，深度随磁盘数量而定，但是最小深度为 1 个扇区。这样的话，每个 Segment 的大小一般就是 1 个扇区或者几个扇区的容量。以图 4.10 的例子来看，有 4 块数据盘和 1 块校验盘。每个 Segment 也就是图中的一个 Block Portion，假如为 2 个扇区大小，就是 1KB，则整个条带的数据部分大小为 4KB。如果一个 Segment 大小为 8 个扇区，即 4KB，则整个条带大小为 16KB。

例解 RAID 3 的作用机制

我们还是用一个例子来说明 RAID 3 的作用机制。一个 4 块数据盘和 1 块校验盘的 RAID 3 系统，Segment SIZE 为 2 个扇区大小(1KB)，条带长度为 4KB。

RAID 3 控制器接收到了这么一个 IO：写入 初始扇区 10000 长度 8，即总数据量为 8×512 字节=4KB。

控制器先定位 LBA10000 所对应的真实物理 LBA，假如 LBA10000 恰好在第一个条带的第一个 Segment 的第一个扇区上，那么控制器将这个 IO 数据里的第 1、2 个 512 字节写入这个扇区。

同一时刻，第 3、4 个 512 字节会被同时写入这个条带的第二个 Segment 中的两个扇区，其后的数据同样被写入第 3、4 个 Segment 中，此时恰好是 4KB 的数据量。也就是说这 4KB 的 IO 数据同时被分散写入了 4 块磁盘，每块磁盘写入了两个扇区，也就是一个 Segment。他们是并行写入的，包括校验盘也是并行写入的，所以 RAID 3 的校验盘没有瓶颈，但是有延迟，因为增加了计算校验的开销。

但现代控制器一般都使用专用的 XOR 硬件电路而不是 CPU 来计算 XOR，这样就使得延迟降到最低。上面那个情况是 IO SIZE 刚好等于一个条带大小的时候，如果 IO SIZE 小于一个条带大小呢？

我们接着分析，还是刚才那个环境，此时控制器接收到 IO 大小为 2KB 的写入请求，也就是 4 个连续扇区，那么控制器就只能同时写入两个磁盘了，因为每个盘上的 Segment 是 2 个扇区，也只能得到两倍的单盘传输速率。同时为了更新校验块，写惩罚也出现了。但是如果同时有个 IO 需要用到另外两块盘，那么恰好可以和当前的 IO 合并起来，这样就可以并发 IO，这种相邻的 IO 一般都是事务性的连续 IO。

再看看 IO SIZE 大于条带长度的情况。还是那个环境，控制器收到的 IO SIZE 为 16KB。则控制器一次所能并行写入的是 4KB，这 16KB 就需要分 4 批来写入 4 个条带。其实这里的分 4 批写入，不是先后写入，而是同时写入，也就是这 16KB 中的第 1、5、9、13KB 将由控制器连续写入磁盘 1，第 2、6、10、14KB，连续写入磁盘 2，依此类推。直到 16KB 数据全部写完，是并行一次写完。这样校验盘也可以一次性计算校验值并且和数据一同并行写入，而不是“分批”。

通过比较，我们发现，与其使 IO SIZE 小于一个条带的大小，从而空闲出一些磁盘，不如使 IO SIZE 大于或者等于条带大小，取消磁盘空余。因为上层 IO SIZE 是不受控的，控制器说了不算，但是条带大小是控制器说了算的。所以如果将条带大小减到很小，比如 2 个扇区、一个扇区，则每次上层 IO 一般情况下都会占用所有磁盘进行并发传输。这样就可以提供和 RAID 2 一样的传输速度，并避免 RAID 2 的诸多缺点。RAID 3 和 RAID 2 一样不能并发 IO， 因为一个 IO 要占用全部盘，就算 IO SIZE 小于 Stripe SIZE，因为校验盘的独享也不

能并发 IO。



一般来说，RAID3 的条带长度 = 文件系统块大小。这样，就不会产生条带不对齐的现象，从而避免产生碎片。

虽然纯 RAID 3 系统不能并发 IO，但是可以通过巧妙的设计，形成 RAID 30 系统。如果文件系统块为 4KB，则使用 8 块数据盘+2 块校验盘做成的 RAID 30 系统，便可以并发 2 个 IO 了。表 4.4 比较了 RAID 3 系统与单盘的性能。

表 4.4 RAID 3 系统相对单盘的 IO 对比

RAID 3 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE 大于 Stripe SIZE	不支持	不支持	提升极小	提升了 N 倍	不支持	不支持	提升极小	提升了 N 倍
IO SIZE 小于 Stripe SIZE	不支持	事务性 IO 可并发，提升并发系数倍	提升极小	提升了 N×IO SIZE/Stripe SIZE 倍	不支持	事务性 IO 可并发，提升并发系数倍	提升极小	提升了 N×IO SIZE/Stripe SIZE 倍

注：N=组成 RAID 3 的数据磁盘数量。和 RAID 2 相同，事务性连续 IO 可能并发。

和 RAID 2 一样，RAID 3 同样也是最适合连续大块 IO 的环境，但是它比 RAID 2 成本更低，更容易部署。

不管任何形式的 RAID，只要是面对随机 IO，其性能与单盘比都没有大的优势，因为 RAID 所做的只是提高传输速率、并发 IO 和容错。随机 IO 只能靠降低单个物理磁盘的寻道时间来解决。而 RAID 不能优化寻道时间。所以对于随机 IO，RAID 3 也同样没有优势。

而对于连续 IO，因为寻道时间的影响因素可以忽略，RAID 3 最拿手了。因为像 RAID 2 一样，RAID 3 可以大大加快数据传输速率，因为它是多盘并发读写。所以理论上可以相当于单盘提高 N 倍的速率。

4.2.5 RAID 4 技术详析

图 4.11 所示为一个 RAID 4 系统的条带布局图。

不管是 RAID 2 还是 RAID 3，它们都是为了大大提高数据传输率而设计，而不能并发 IO。诸如数据库等应用的特点就是高频率随机 IO 读。想提高这种环境的 IOPS，根据公式：**IOPS=IO 并发系数/(寻道时间+数据传输时间)**，随机读导致寻道时间增大，靠提高传输速率已经不是办法。所以观察这个公式，想在随机 IO 频发的环境中提高 IOPS，惟一能够做的要么用高性能的磁盘(即平均寻道时间短的磁盘)，要么提高 IO 并发系数。不能并发 IO 的，想办法让它并发 IO。并发系数小的，想办法提高系数。

	Extent 0	Extent 1	Extent 2	Extent 3	Extent 4
Stripe 0 (Depth)	Strip (0,0)	Strip (1,0)	Strip (2,0)	Strip (3,0)	Strip (4,0)
	Parity (0,0,0)	Data Block 0	Data Block 4	Data Block 8	Data Block 12
	Parity (1,0,0)	Data Block 1	Data Block 5	Data Block 9	Data Block 13
	Parity (2,0,0)	Data Block 2	Data Block 6	Data Block 10	Data Block 14
	Parity (3,0,0)	Data Block 3	Data Block 7	Data Block 11	Data Block 15
Extent Stripe Length					
Stripe 1 (Depth)	Strip (0,1)	Strip (1,1)	Strip (2,1)	Strip (3,1)	Strip (4,1)
	Parity (0,0,1)	Data Block 16	Data Block 20	Data Block 24	Data Block 28
	Parity (1,0,1)	Data Block 17	Data Block 21	Data Block 25	Data Block 29
	Parity (2,0,1)	Data Block 18	Data Block 22	Data Block 26	Data Block 30
	Parity (3,0,1)	Data Block 19	Data Block 23	Data Block 27	Data Block 31
Stripe 2 (Depth)	Strip (0,2)	Strip (1,2)	Strip (2,2)	Strip (3,2)	Strip (4,2)
	Parity (0,0,2)	Data Block 32	Data Block 36	Data Block 40	Data Block 44
	Parity (1,0,2)	Data Block 33	Data Block 37	Data Block 41	Data Block 45
	Parity (2,0,2)	Data Block 34	Data Block 38	Data Block 42	Data Block 46
	Parity (3,0,2)	Data Block 35	Data Block 39	Data Block 43	Data Block 47
	⋮	⋮	⋮	⋮	⋮

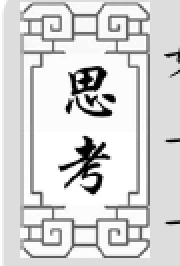
图4.11　RAID 4 系统示意图

思考

在 RAID 3 的基础上，RAID 4 被发展起来。我们分析 RAID 3 的性能的时候，曾经提到过一种情况，就是 IO SIZE 小于 Stripe SIZE 的时候，此时有磁盘处于空闲状态。如果抓住这个现象，同时让队列中的其他 IO 来利用这些空闲的磁盘，岂不是正好达到并发 IO 的效果了么？所以 RAID 4 将一个 Segment 的大小做得比较大，以至于平均 IO SIZE 总是小于 Stripe SIZE，这样就能保证每个 IO 少占用磁盘，甚至一个 IO 只占用一个磁盘。

是的，这个思想对于读 IO 是对路子的，但是对于写 IO 的话，有一个很难克服的问题，那就是校验盘的争用。考虑一下这样一种情况，在 4 块数据盘和 1 块校验盘组成的 RAID 4 系统中，某时刻一个 IO 占用了前两块盘和校验盘，此时虽然后两块是空闲的，可以同时接受新的 IO 请求。但是接受了新的 IO 请求，则新 IO 请求同样也要使用校验盘。由于一块物理磁盘不能同时处理多个 IO，所以新 IO 仍然要等旧 IO 写完后，才能写入校验。这样就和顺序 IO 没区别了。数据盘可并发而校验盘不可并发，这样不能实现写 IO 并发。

如果仅仅根据争用校验盘来下结论说 RAID 4 不支持并发 IO，也是片面的。我们可以设想这样一种情形，某时刻一个 IO 只占用了全部磁盘的几块盘，另一些磁盘空闲。如果此时让队列中下一个 IO 等待的话，那么当然不可能实现并发 IO。



如果队列中有这样一个 IO，它需要更新的 LBA 目标和正在进行的 IO 恰好在同一条带上，并且处于空闲磁盘，还不冲突，那么此时我们就可以让这个 IO 也搭一下正在进行的 IO 的顺风车。反正都是要更新这个条带的校验 Segment，与其两个 IO 先后更新，不如让它们同时更新各自的数据 Segment，而控制器负责计算本条带的校验块。这样就完美的达到了 IO 并发。

但是，这种情况遇到的几率真是小之又小。即便如此，控制器如果可以对队列中的 IO 目标 LBA 进行扫描，将目标处于同一条带的 IO，让其并发写入，这就多少类似 NCQ 技术了。

但是如果组合动作在上层就已经算好了，人为的创造并发条件，主动去合并可以并发的，合并号之后再下发给下层，那么事务并发 IO 的几率将大大增加，而不是靠底层碰运气来实现，不过此时称为“并发事务”更为合适。

所谓“上层”是什么呢？上层就是一级磁盘控制器驱动程序的上层，也就是文件系统层。文件系统管理着底层磁盘，决定数据写往磁盘或者虚拟卷上的哪些块。所以完全可以在文件系统这个层次上，将两个不同事务的 IO 写操作，尽量放到相同的条带上。比如一个条带大小为 16KB，可以前 8KB 放一个 IO 的数据，后 8KB 放另一个 IO 的数据，这两个 IO 在经过文件系统的计算之后，经由磁盘控制器驱动程序，向磁盘发出同时写入整个条带的操作，这样就构成了整条写、如果实在不能占满整条，那么也应该尽量达成重构写模式，这样不但并发了 IO，还使得写效率增加。



这种在文件系统专门为 RAID 4 做出优化的方案，最典型的就是 NetApp 公司著名的 WAFL 文件系统。WAFL 文件系统的设计方式确保了能够最大限度地实现整条写操作。

图 4.12 右半部对比显示了 WAFL 如何分配同样的数据块，从而使得 RAID 4 更加有效。WAFL 总是把可以合并写入的数据块尽量同时写到一个条带中，以消除写惩罚，增加 IO 并发系数。相对于 WAFL，左边的 FFS(普通文件系统)由于对 RAID 4 没有感知，产生的 IO 不适合 RAID 4 的机制，从而被零散的分配到了 6 个独立的条带，因此致使 6 个校验盘块需要更新，而只能顺序的进行，因为校验盘不可并发。而右边的 WAFL 仅仅使用 3 道条带，只有 3 个校验块需要更新，从而大大提高了性能。表 4.5 比较了 RAID 4 系统与单盘的性能。

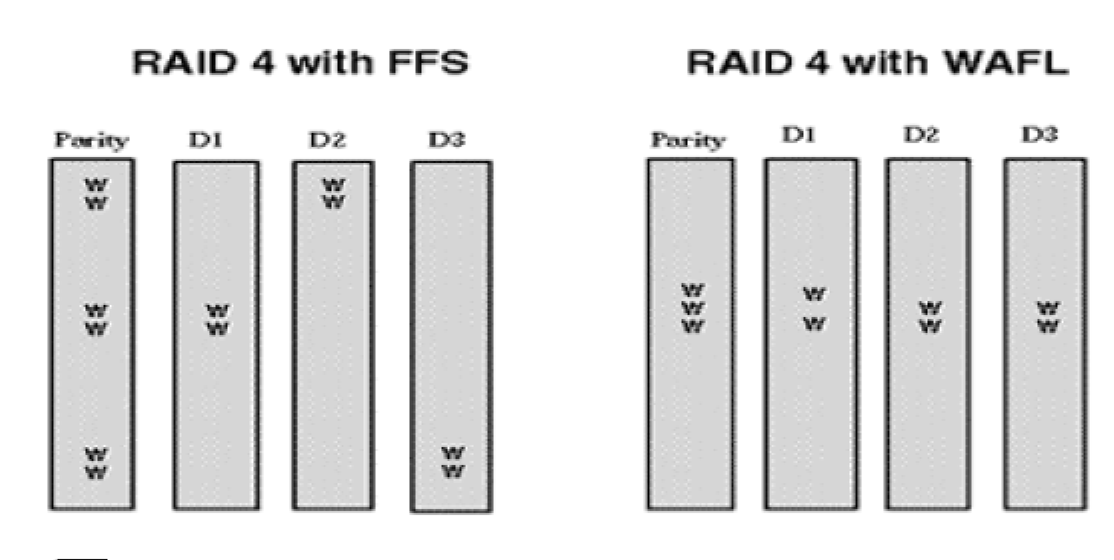


图4.12 普通文件系统与 WAFL 文件系统的对比

表 4.5 RAID 4 系统相对单盘的 IO 对比

RAID 4 IOPS	读				写			
	特别优化的并发 IO		顺序 IO		特别优化的并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE/Stripe SIZE 较大	冲突	冲突	提升极小	提升了 N 倍	冲突	冲突	没有提升	提升了 N 倍
IO SIZE/Stripe SIZE 较小	提升极小	提升并发系数×N 倍	几乎没有提升	几乎没有提升	提升并发系数倍	提升并发系数×N 倍	性能降低	性能降低

注：N 为 RAID 4 数据盘数量。IO SIZE/Stripe SIZE 太大则并发 IO 几率很小。



如果 IO SIZE/Stripe SIZE 的值太小，那么顺序 IO 读不管是连续还是随机 IO 几乎都没有提升。顺序 IO 写性能下降，是因为 IO SIZE 很小，又是顺序 IO，只能进行读改写，性能会降低不少。

所以，如果要使用 RAID 4，不进行特别优化是不行的，至少要让它可以进行并发 IO。观察表 4.5 可知，并发 IO 模式下性能都有所提升。然而如果要优化到并发几率很高，实在不容易。目前只有 NetApp 的 WAFL 文件系统还在使用 RAID 4，其他产品均未见使用。RAID 4 面临淘汰，取而代之的是拥有高盲并发几率的 RAID 5 系统。所谓盲并发几率，就是说上层不必感知下层的结构，即可增加并发系数。

4.2.6 RAID 5 技术详析

图 4.13 为一个 RAID 5 系统的条带布局图。

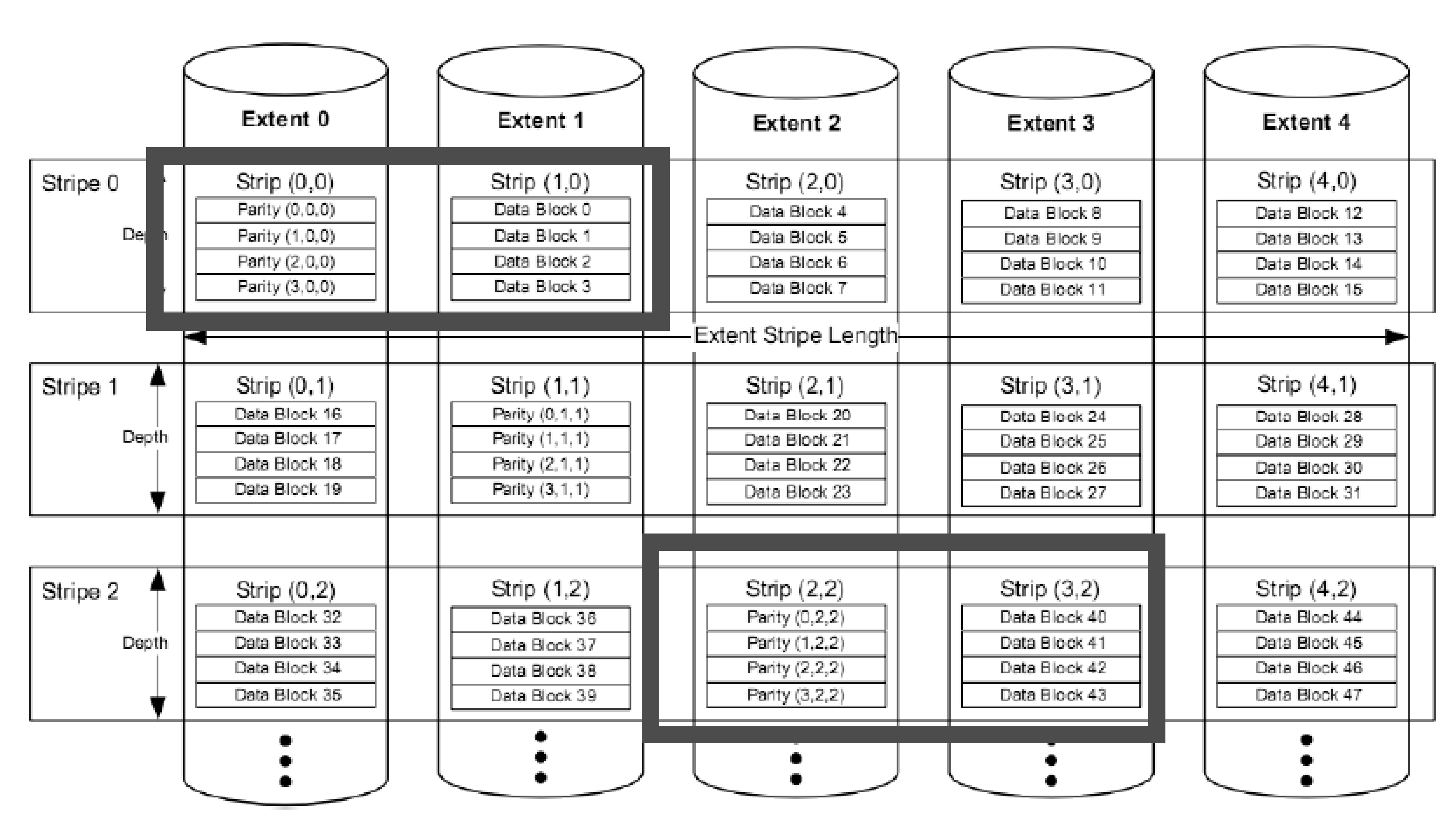


图4.13 RAID 5 系统示意图

整条写、重构写与读改写

- 整条写(Full-Stripee Write)：整条写需要修改奇偶校验群组中所有的条带单元，因此新的奇偶校验值可以根据所有新的条带数据计算得到，不需要额外的读、写操作。因此，整条写是最有效的写类型。整条写的例子，如 RAID 2、RAID 3。它们每次 IO 总是几乎能保证占用所有盘，因此每个条带上的每个 Segment 都被写更新，所以控制器可以直接利用这些更新的数据计算出校验数据之后，在数据被写入数据盘的同时，将计算好的校验信息写入校验盘。
- 重构写(Reconstruct Write)：如果要写入的磁盘数目超过阵列磁盘数目的一半，可采取重构写方式。在重构写中，从这个条带中不需要修改的 Segment 中读取原来的数据，再和本条带中所有需要修改的 Segment 上的新数据计算奇偶校验值，并将新的 Segment 数据和没有更改过的 Segment 数据以及新的奇偶校验值一并写入。显然，重构写要牵涉更多的 I/O 操作，因此效率比整条写低。重构写的例子，比如在 RAID 4 中，如果数据盘为 8 块，某时刻一个 IO 只更新了一个条带的 6 个

Segment，剩余两个没有更新。在重构写模式下，会将没有被更新的两个 Segment 的数据读出，和需要更新的前 6 个 Segment 的数据计算出校验数据，然后将这 8 个 Segment 连同校验数据一并写入磁盘。可以看出，这个操作只是多出了读两个 Segment 中数据的操作和写两个 segment 的操作，但是写的时候几乎不产生延迟开销，因为是宏观同时写入。

- 读改写(Read-Modify Write)：如果要写入的磁盘数目不足阵列磁盘数目的一半，可采取读改写方式。读改写过程是：先从需要修改的 Segment 上读取旧的数据，再从条带上读取旧的奇偶校验值；根据旧数据、旧校验值和需要修改的 Segment 上的新数据计算出这个条带上的新的校验值；最后写入新的数据和新的奇偶校验值。这个过程中包含读取、修改和写入的一个循环周期，因此称为读改写。读改写计算新校验值的公式为：**新数据的校验数据=(老数据 EOR 新数据) EOR 老校验数据**。如果待更新的 Segment 已经超过了条带中总 Segment 数量的一半，则此时不适合用读改写，因为读改写需要读出这些 Segment 中的数据和校验数据。而如果采用重构写，只需要读取剩余不准备更新数据的 Segment 中的数据即可，而后者数量比前者要少。所以超过一半用重构写，不到一半用读改写。整条更新就用整条写。

写效率排列为整条写>重构写>读改写。

图 4.14 是 RAID 5 系统的三种写模式示意图。

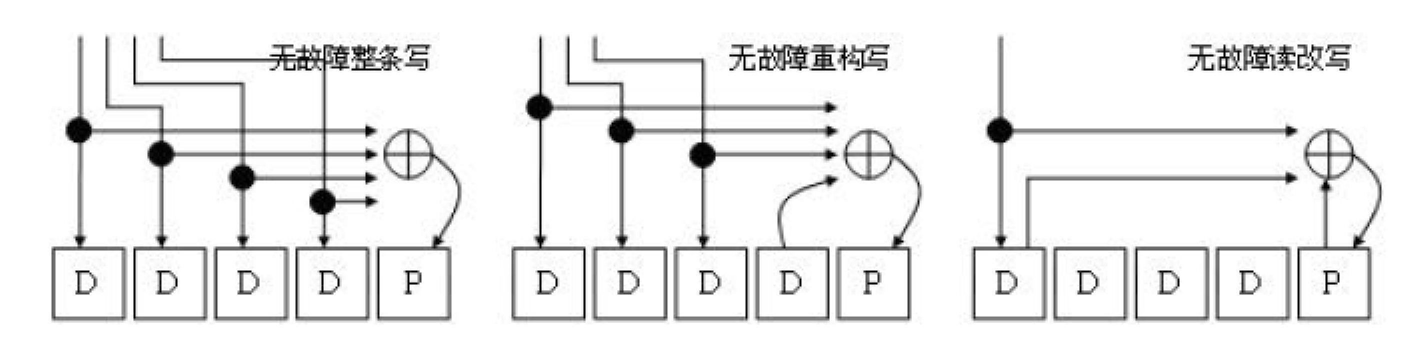


图 4.14 RAID 5 写模式示意图

为了解决 RAID 4 系统不能并发 IO 的窘境，RAID 5 相应而出。RAID 4 并发困难是因为它的校验盘争用的问题，如果能找到一种机制可以有效地解决这个问题，则实现并发就会非常容易。RAID 5 恰恰解决了校验盘争用这个问题。RAID 5 采用分布式校验盘的做法，将校验盘打散在 RAID 组中的每块磁盘上。如图 4.14 所示，每个条带都有一个校验 Segment，但是不同条带中位置不同，在相邻条带之间循环分布。为了保证并发 IO，RAID 5 同样将条带大小做得较大，以保证每次 IO 数据不会占满整个条带，造成队列中其他 IO 的等待。所以，RAID 5 要保证高并发率，一旦某时刻没有成功进行并发，则这个 IO 几乎就是读改写模式，所以 RAID 5 拥有较高的写惩罚。

但是在随机写 IO 频发的环境下，由于频发的随机 IO 提高了潜在的并发几率，如果碰巧并发的 IO 同处一个条带，还可以降低写惩罚的几率。这样，RAID 5 系统面对频发的随机写 IO，其 IOPS 下降趋势比其他 RAID 类型要平缓一些。

来分析一下 RAID 5 具体的作用机制。以图 4.14 的环境为例，条带大小 80KB，每个 Segment 大小 16KB。

1】 某一时刻，上层产生一个写 IO：写入 初始扇区 10000 长度 8，即写入 4KB 的数据。控制器收到这个 IO 之后，首先定位真实 LBA 地址，假设定位到了第 1 个条带

的第 2 个 Segment(位于图中的磁盘 2)的第 1 个扇区(仅仅是假设)，则控制器首先对这个 Segment 所在的磁盘发起 IO 写请求，读取这 8 个扇区中原来的数据到 Cache。

2】 与此同时，控制器也向这个条带的校验 Segment 所在的磁盘(即图中的磁盘 1)发起 IO 读请求，读出对应的校验扇区数据并保存到 Cache。

3】 利用 XOR 校验电路来计算新的校验数据，公式为：**新数据的校验数据=(老数据 EOR 新数据) EOR 老校验数据**。现在 Cache 中存在：老数据、新数据、老校验数据和新校验数据。

4】 控制器立即再次向相应的磁盘同时发起 IO 写请求，将新数据写入数据 Segment，将新校验数据写入校验 Segment，并删除老数据和老校验数据。

在上述过程中，这个 IO 占用的始终只有 1、2 两块盘，因为所要更新的数据 Segment 对应的校验 Segment 位于 1 盘，自始至终都没有用到其他任何磁盘。如果此时队列中有这么一个 IO，它的 LBA 初始目标假如位于图中下方红框所示的数据 Segment 中(磁盘 4)，IO 长度也不超过 Segment 的大小。而这个条带对应的校验 Segment 位于磁盘 3 上。这两块盘未被其他任何 IO 占用，所以此时控制器就可以并发的处理这个 IO 和上方红框所示的 IO，达到并发。

RAID 5 相对于经过特别优化的 RAID 4 来说，在底层就实现了并发，可以脱离文件系统的干预。任何文件系统的 IO 都可以实现较高的并发几率，又称为盲并发。而不像基于 WAFL 文件系统的 RAID 4，需要在文件系统上规划计算出并发环境。然而就效率来说，仍然是 WAFL 拥有更高的并发系数，因为毕竟 WAFL 是靠主动创造并发，而 RAID 5 却是做好了陷阱等人往里跳，抓着一个是一个。

RAID 5 磁盘数量越多，可并发的几率就越大。

表 4.6 RAID 5 系统相对单盘的 IO 对比

RAID 5 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE 近似 Stripe SIZE	不支持	不支持	提升极小	提升了 N 倍	不支持	不支持	提升极小	提升了 N 倍
IO SIZE 大于 Segment SIZE 且重构写	提升并发系数倍	提升并发系数×N 倍	几乎没有提升	提升了 IO SIZE/Segment SIZE 倍	提升并发系数倍	提升并发系数倍	性能下降	提升极小
IO SIZE 小于 Segment SIZE 且读改写	提升并发系数倍	提升并发系数×N 倍	提升极小	没有提升	提升并发系数倍	提升并发系数×N 倍	性能下降	性能下降

注：RAID 5 最适合小块 IO。并发 IO 的情况下，性能都较单盘有所提升。

图 4.15 为一个 RIAD5E 系统的条带布局图。

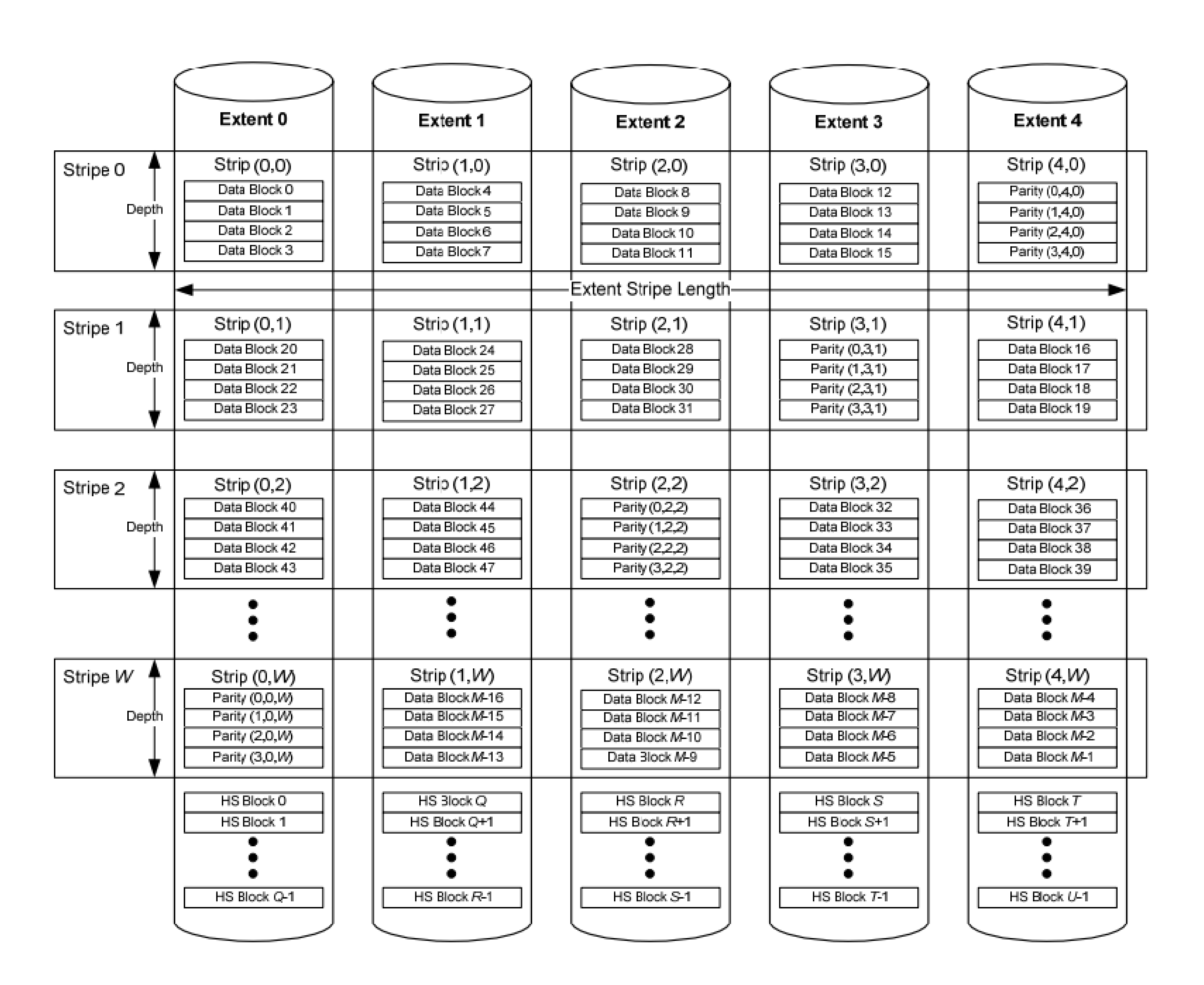


图4.15 RAID 5E 示意图(HS 代表 HotSpare)

图 4.16 则为一个 RAID 5EE 系统的条带布局图。

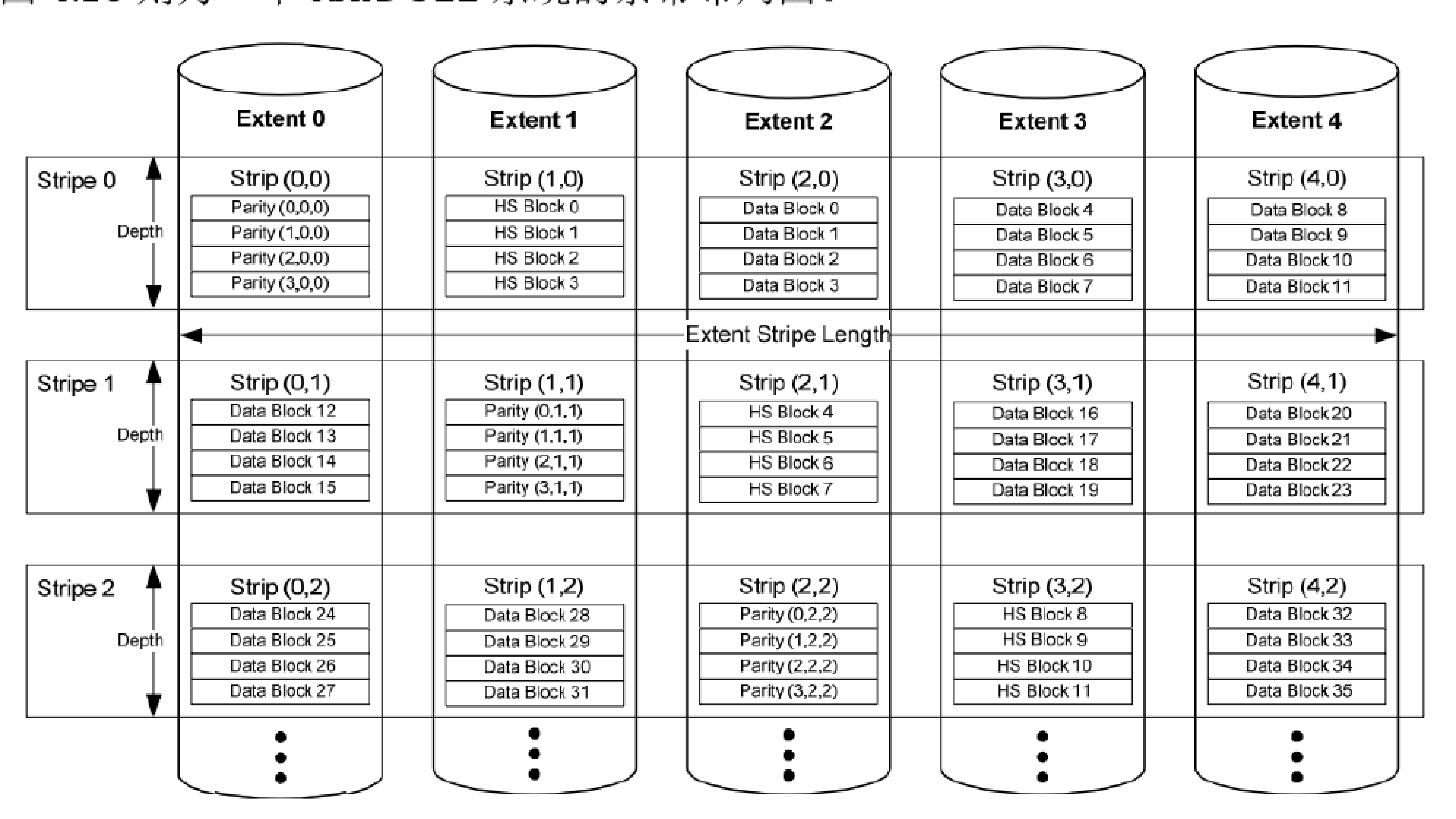


图4.16 RAID 5EE 示意图

4.2.7 RAID 6 技术详析

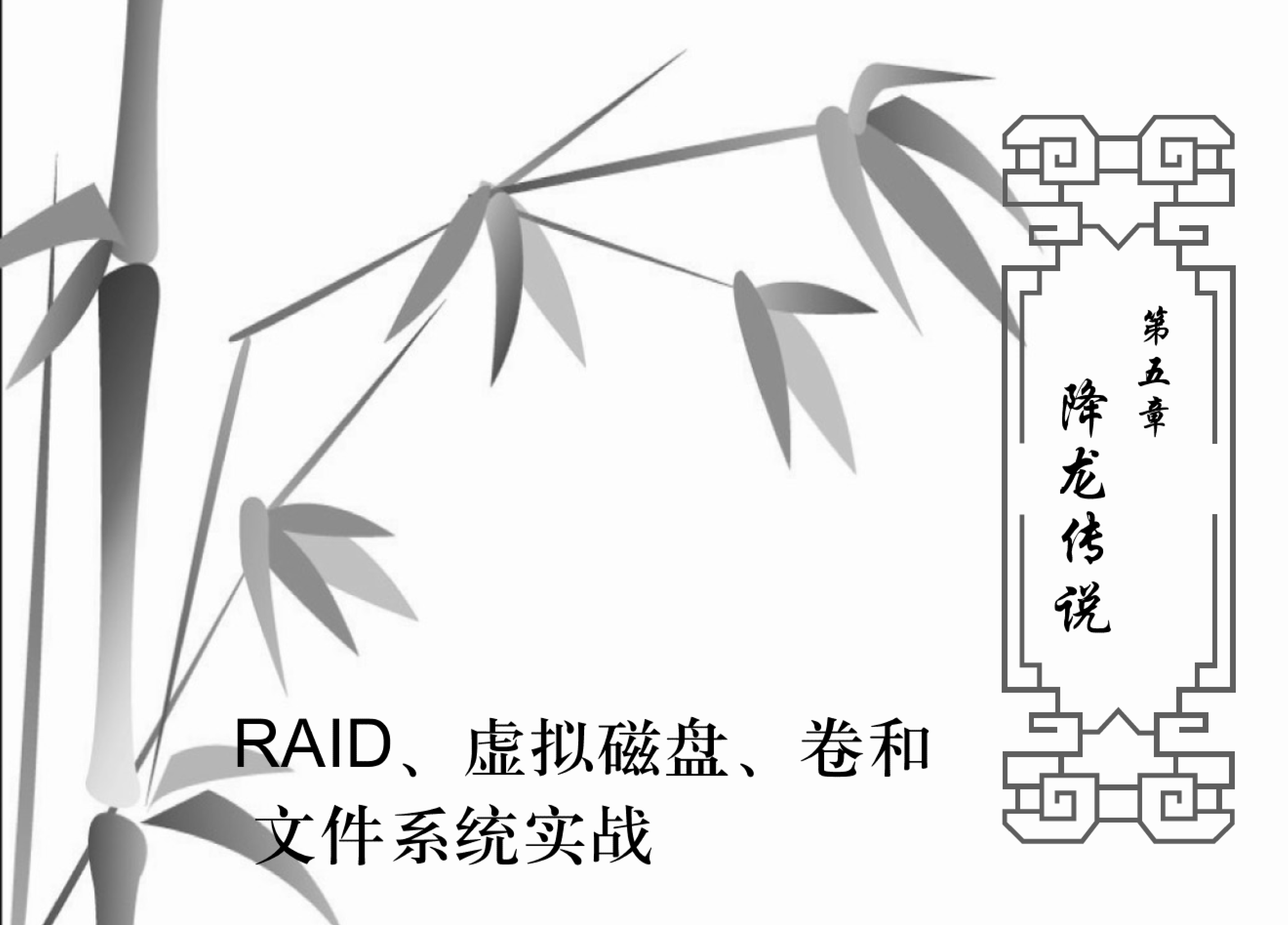
图 4.17 为一个 RAID 6 系统的条带布局图。

图4.17 RAID 6 系统示意图

RAID 6 之前的任何 RAID 级别，最多能保障在坏掉一块盘的时候，数据仍然可以访问。如果同时坏掉两块盘，则数据将会丢失。为了增加 RAID 5 的保险系数，RAID 6 被创立了。RAID 6 比 RAID 5 多增加了一块校验盘，也是分布打散在每块盘上，只不过是用另一个方程式来计算新的校验数据。这样，RAID 6 同时在一个条带上保存了两份数学上不相关的校验数据，这样能够保证同时坏两块盘的情况下，数据依然可以通过联立这两个数学关系等式来求出丢失的数据。RAID 6 与 RAID 5 相比，在写的时候会同时读取或者写入额外的一份校验数据。不过由于是并行同时操作，所以不比 RAID 5 慢多少。其他特性则和 RAID 5 类似。

表 4.7 RAID 6 系统相对单盘的 IO 对比

RAID 6 IOPS	读				写			
	并发 IO		顺序 IO		并发 IO		顺序 IO	
	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO	随机 IO	连续 IO
IO SIZE 近似 Stripe SIZE	不支持	不支持	提升极小	提升了 N 倍	不支持	不支持	提升极小	提升了 N 倍
IO SIZE 大于 Segment SIZE 重构写	提升并发系数倍	提升并发系数×N 倍	几乎没有提升	提升了 IO SIZE/ Segment SIZE 倍	提升并发系数倍	提升并发系数×N 倍	性能下降	提升极小
IO SIZE 小于 Segment SIZE 读改写	提升并发系数倍	提升并发系数倍	提升极小	没有提升	提升并发系数倍	提升并发系数倍	性能下降	性能下降



RAID、虚拟磁盘、卷和文件系统实战

- RAID 卡
- 软 RAID
- 虚拟磁盘
- 卷
- 文件系统

七星大侠将七星北斗阵式永传于世，虽然其思想博大精深，但是这个阵式并没有给出如何去具体地实现这七种阵式。但没有关系，有了正确的思想才能更好地指导实践。人们根据七星北斗的思想，发明了各种各样的 RAID 实现方式。

然而，实现了各种 RAID，许多问题也随之而来，且看人们是怎么运用各种手段来解决这些问题的。

5.1 操作系统中 RAID 的实现和配置

有人直接在主机上编写程序，运行于操作系统底层，将从主机 SCSI 或者 IDE 控制器提交上来的物理磁盘，运用七星北斗的思想，虚拟成各种模式的虚拟磁盘，然后再提交给上层程序接口，如卷管理程序。这些软件通过一个配置工具，让使用者自行选择将哪些磁盘组合起来并形成哪种类型的 RAID。

比如，某台机器上安装了 2 块 IDE 磁盘和 4 块 SCSI 磁盘，IDE 硬盘直接连接到主板集成的 IDE 接口上，SCSI 磁盘则是连接到一块 PCI 接口的 SCSI 卡上。在没有 RAID 程序参与的条件下，系统可以识别到 6 块磁盘，并且经过文件系统格式化之后，挂载到某个盘符或者目录下，供应用程序读写。

安装了 RAID 程序之后，用户通过配置界面，先将两块 IDE 磁盘做成了一个 RAID 0 系统。如果原来每块 IDE 磁盘是 80GB 容量，做成 RAID 0 之后就变成了一块 160GB 容量的"虚拟"磁盘。然后用户又将 4 块 SCSI 盘做了一个 RAID 5 系统，如果原来每块 SCSI 磁盘是 73GB 容量，4 块盘做成 RAID 5 之后虚拟磁盘的容量将约为 3 块盘的容量，即 216GB。

当然，因为 RAID 程序需要使用磁盘上的部分空间来存放一些 RAID 信息，所以实际容量将会变小。经过 RAID 程序的处理之后，这 6 块磁盘最终变成了两块虚拟磁盘。如果是在 Windows 系统中，打开磁盘管理器只能看到 2 块硬盘，一块容量为 160GB(硬盘 1)，另一块容量为 216GB(硬盘 2)。之后，可以对这两块盘进行格式化，比如格式化为 NTFS 文件系统。格式化程序丝毫不会感觉到有多块物理硬盘正在写入数据。

比如，格式化程序某时刻发出命令，向硬盘 1(由两块 IDE 磁盘组成的 RAID 0 虚拟盘)的 LBA 起始地址 10000，长度 128，写入内存起始地址某某的数据。RAID 程序会截获这个命令并做分析，硬盘 1 是一个 RAID 0 系统，那么这块从 LBA10000 开始算起的 128 个扇区的数据，会被 RAID 引擎计算，将逻辑 LBA 对应成物理磁盘的物理 LBA，将对应的数据写入物理磁盘。写入之后，格式化程序会收到成功写入的信号，然后接着做下一次 IO。经过这样的处理，上层程序完全不会知道底层物理磁盘的细节。其他 RAID 形式也都是相同的道理，只不过算法更加复杂而已。但是即使再复杂的算法，经过 CPU 运算，也要比磁盘读写速度快几千几万倍。



为了保证性能，同一个磁盘组只能用相同类型的磁盘，虽然也可以设计成将 IDE 磁盘和 SCSI 磁盘组合成虚拟磁盘，不过除非特殊需要，否则没有这样设计的。

5.1.1 Windows Server 2003 高级磁盘管理

我们以 Windows Server 2003 企业版操作系统为例，示例一下 Windows 是如何在操作系统上用软件来实现 RAID 功能的。

每个例子的环境都是一个具有 5 块物理磁盘的 PC 机，每块磁盘容量为 100MB。

1. 磁盘初始化和转换

1】 新磁盘插入机箱并启动操作系统之后，打开磁盘管理器，Windows 会自动弹出一个配置新磁盘的向导，如图 5.1 所示。

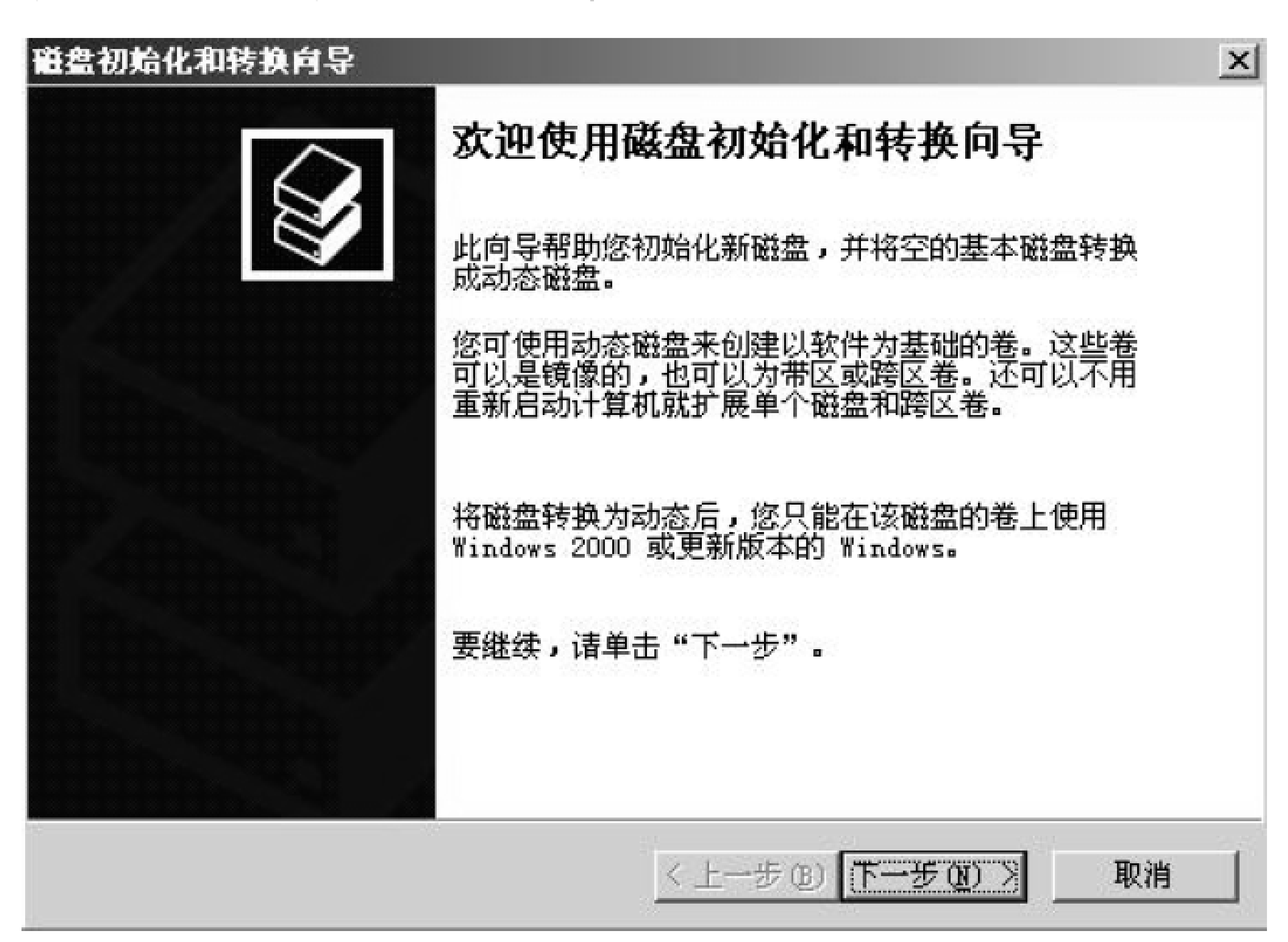


图5.1 初始界面

2】 单击“下一步”按钮，出现图 5.2 所示的对话框。

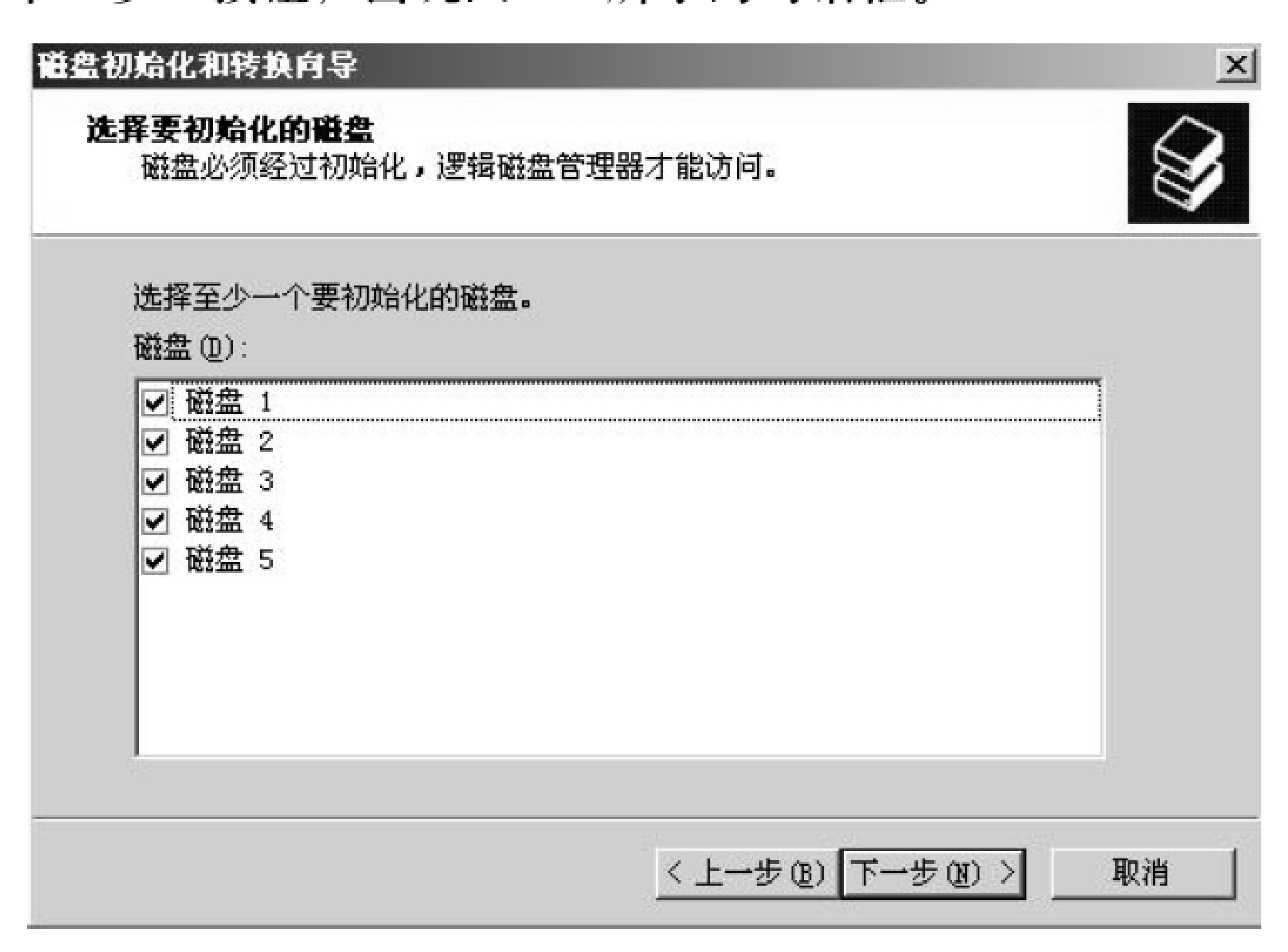


图5.2 选择要初始化的磁盘

3】 单击“下一步”按钮，初始化所有新磁盘，如图 5.3 所示。

4】 单击“下一步”按钮，将所有磁盘转换为动态磁盘，如图 5.4 所示。所谓的动态磁盘就是可以用来做 RAID 以及卷管理的磁盘。

5】 单击“完成”按钮。查看磁盘管理器中的状态，如图 5.5 所示。

我们从图 5.5 中可以看到，磁盘 0 为基本磁盘，同时也是系统所在的磁盘以及启动磁盘。这个磁盘不能对其进行软 RAID 或卷管理操作。



图5.3 选择要转换的磁盘

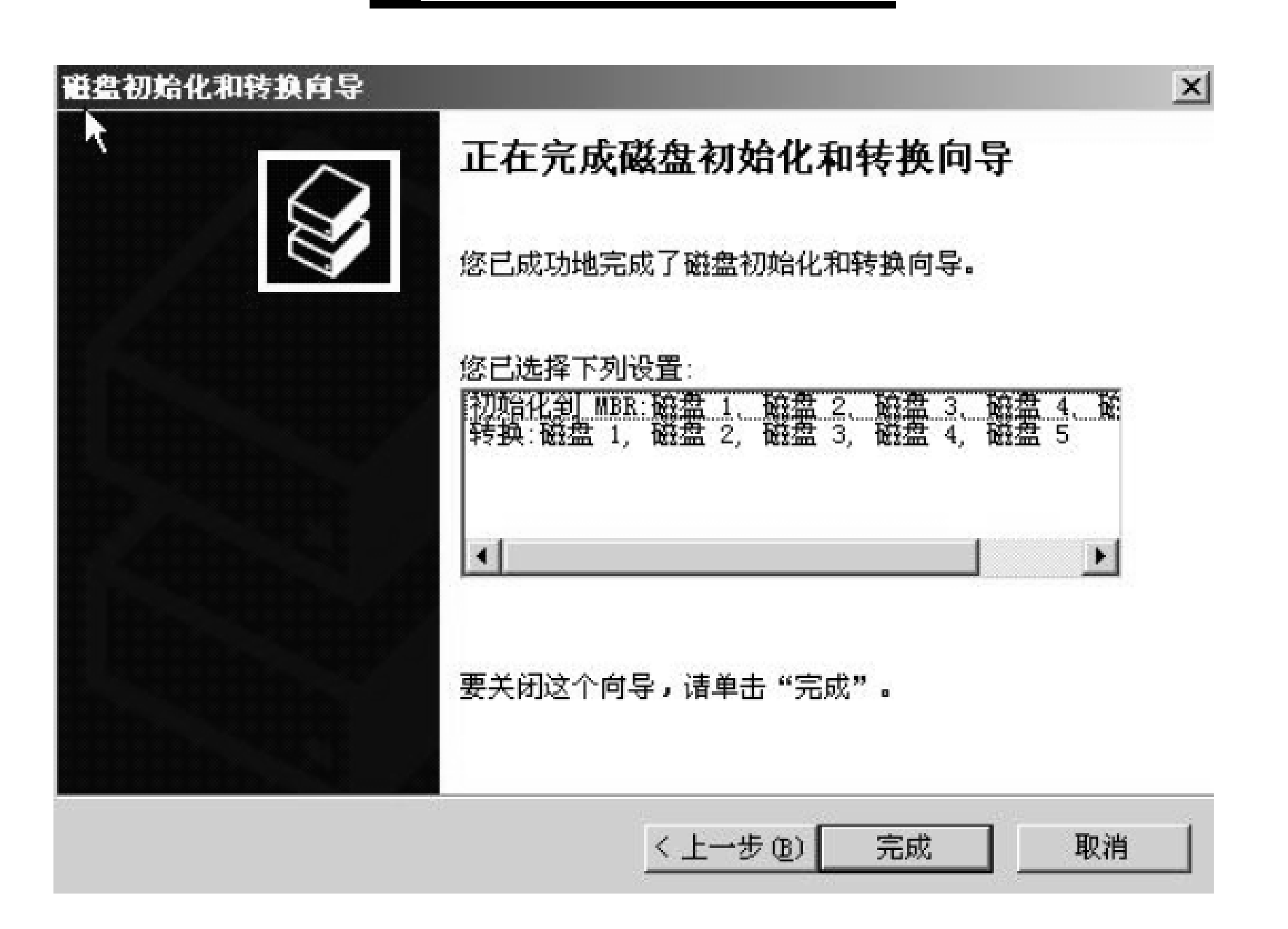


图5.4 初始化磁盘

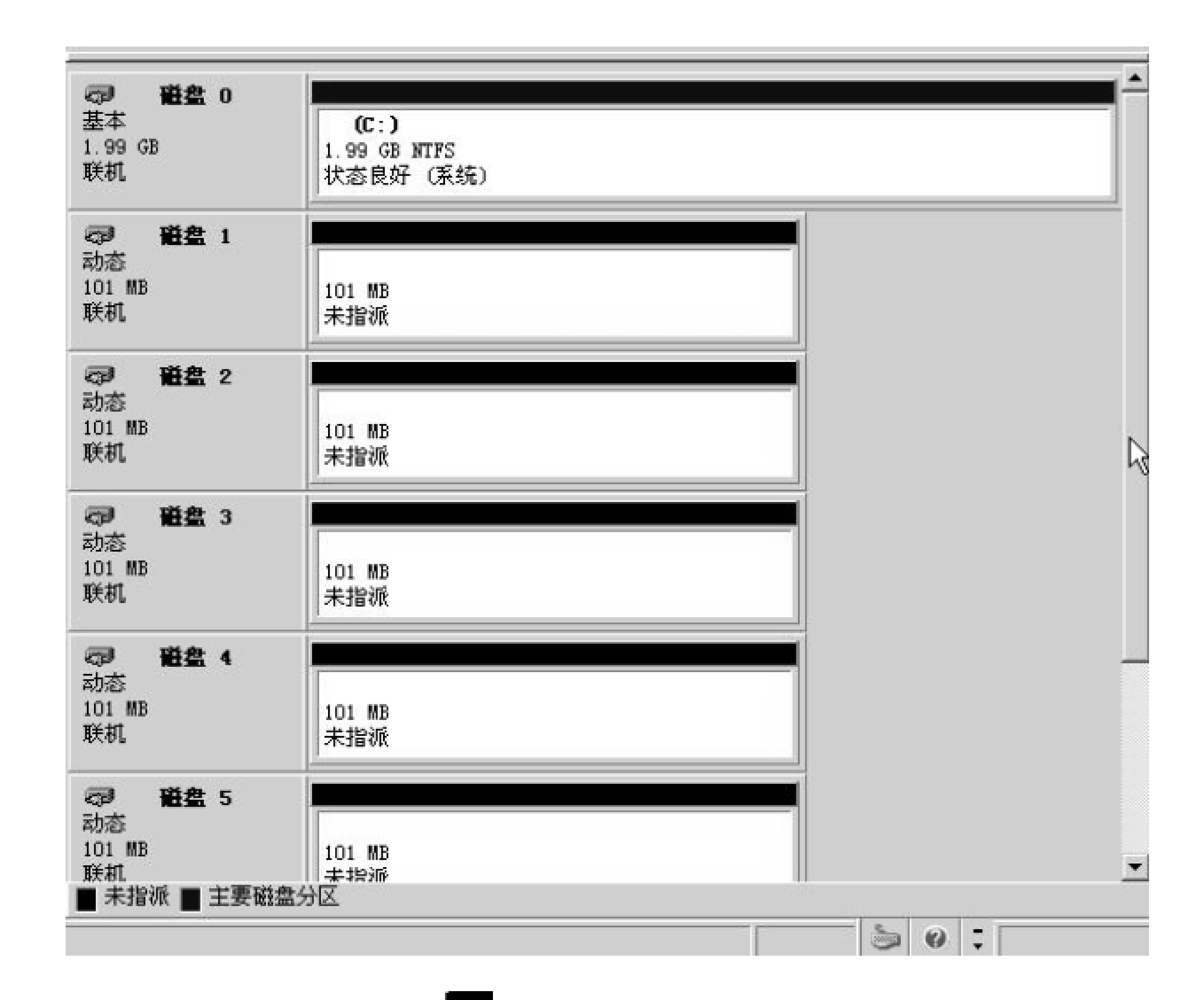


图5.5 磁盘状态

2. 新建卷

1】 在“磁盘 1”上右击，在弹出的快捷菜单中选择“新建卷”命令，如图 5.6 所示，系统弹出“新建卷向导”对话框，以选择要创建的卷的类型，如图 5.7 所示。

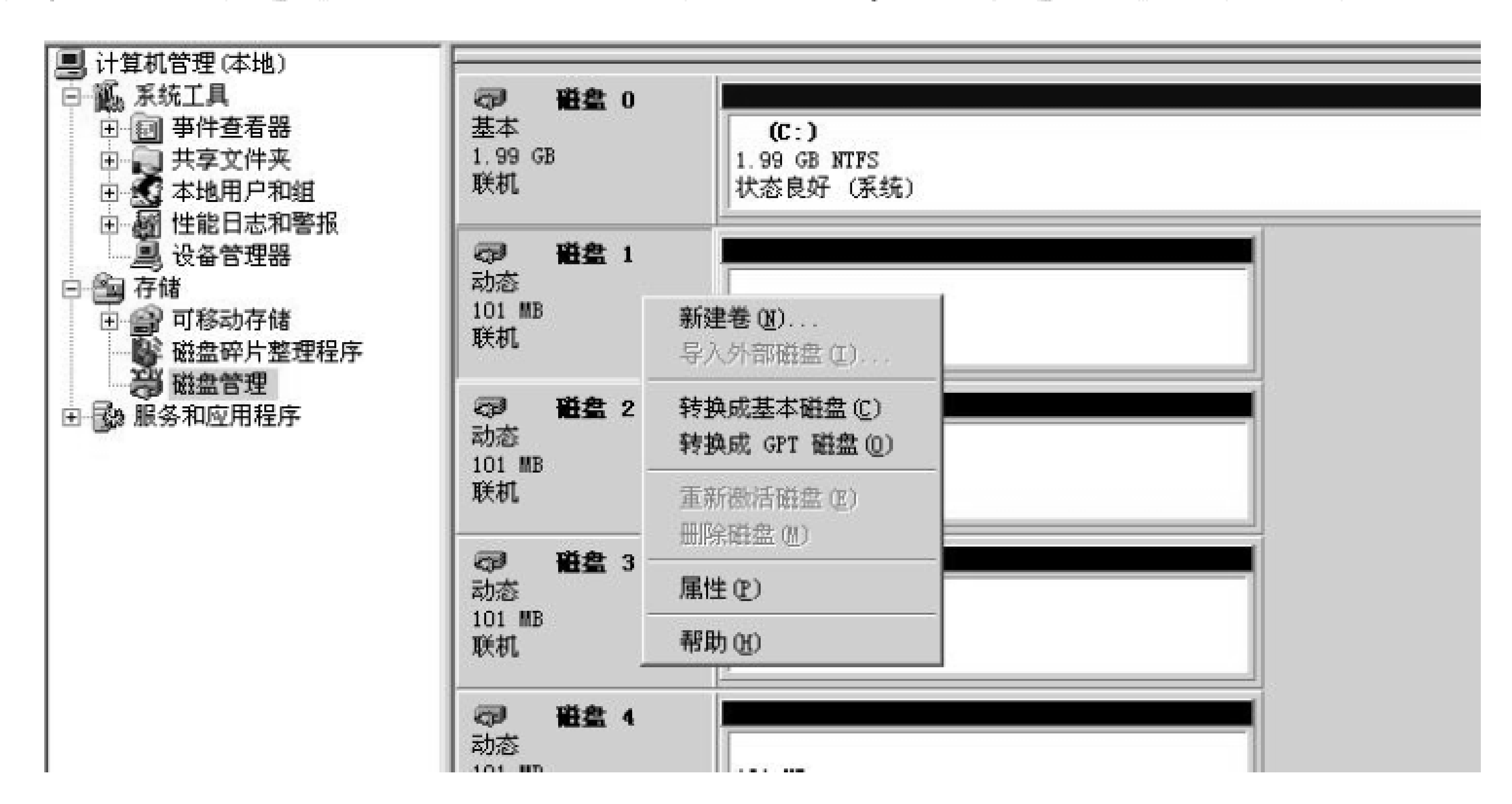


图5.6 选择“新建卷”命令

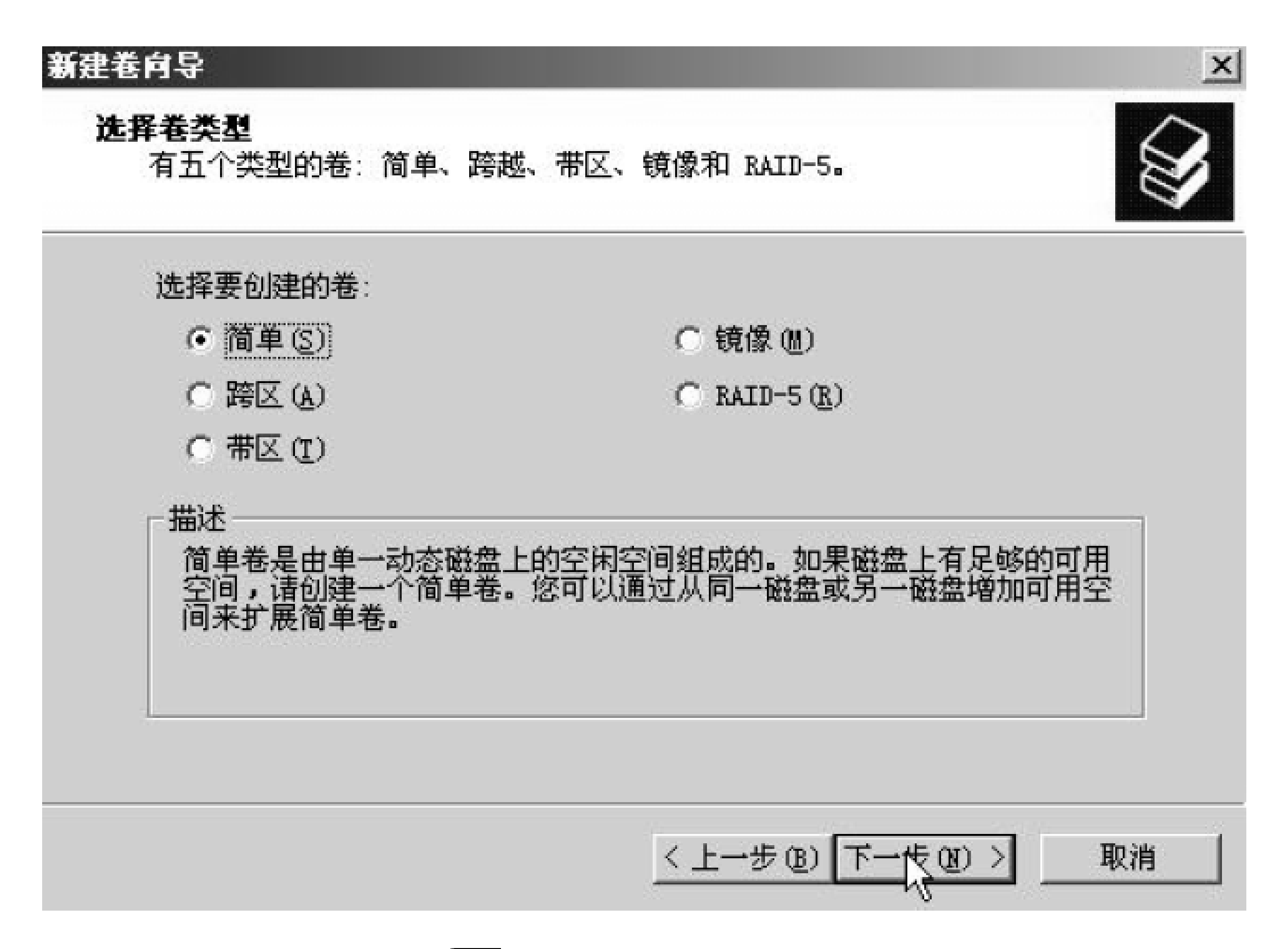


图5.7 选择卷类型

这里有 5 个选项，下面分别来介绍这些选项。

- 简单卷：指卷将按照磁盘的顺序依次分配空间。简单卷与磁盘分区功能类似，卷空间只能在一块磁盘上分配，并且不能交叉或者乱序。
- 跨区卷：跨区卷在简单卷的基础上，可以让一个卷的空间跨越多块物理磁盘。相当于不做条带化的 RAID 0 系统。
- 带区卷：带区卷相当于条带化的 RAID 0 系统。
- 镜像卷：镜像卷相当于 RAID 1 系统。
- RAID-5 卷：毫无疑问，这种方式就是实现一个 RAID 5 卷。

图 5.8 做的是一个大小为 101MB 的简单卷，也就是将物理磁盘 1 全部容量划分给这个卷。可以发现，简单卷只能在一块物理磁盘上划分，图中“添加”按钮是灰色的，证明不能跨越多块磁盘。

我们再来看看跨区卷，如图 5.9 所示。

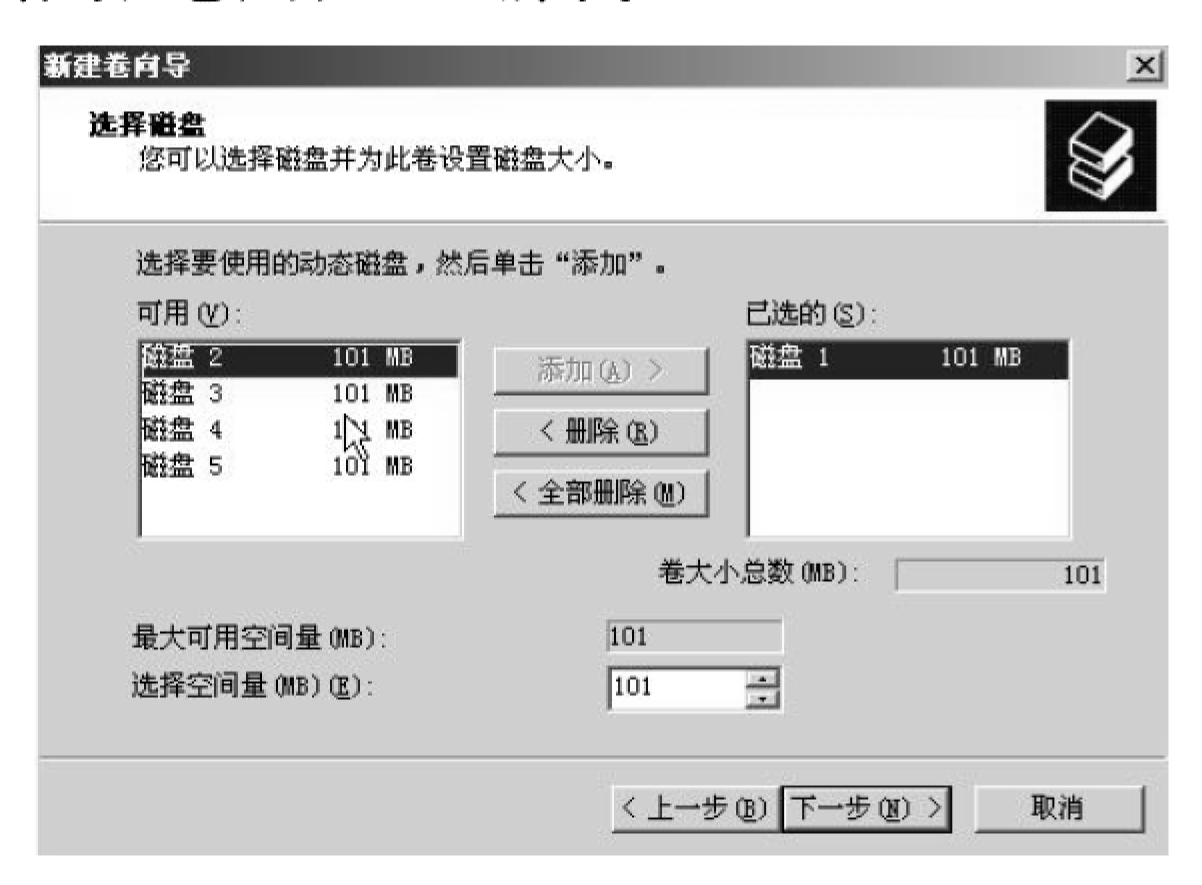


图5.8　划分大小

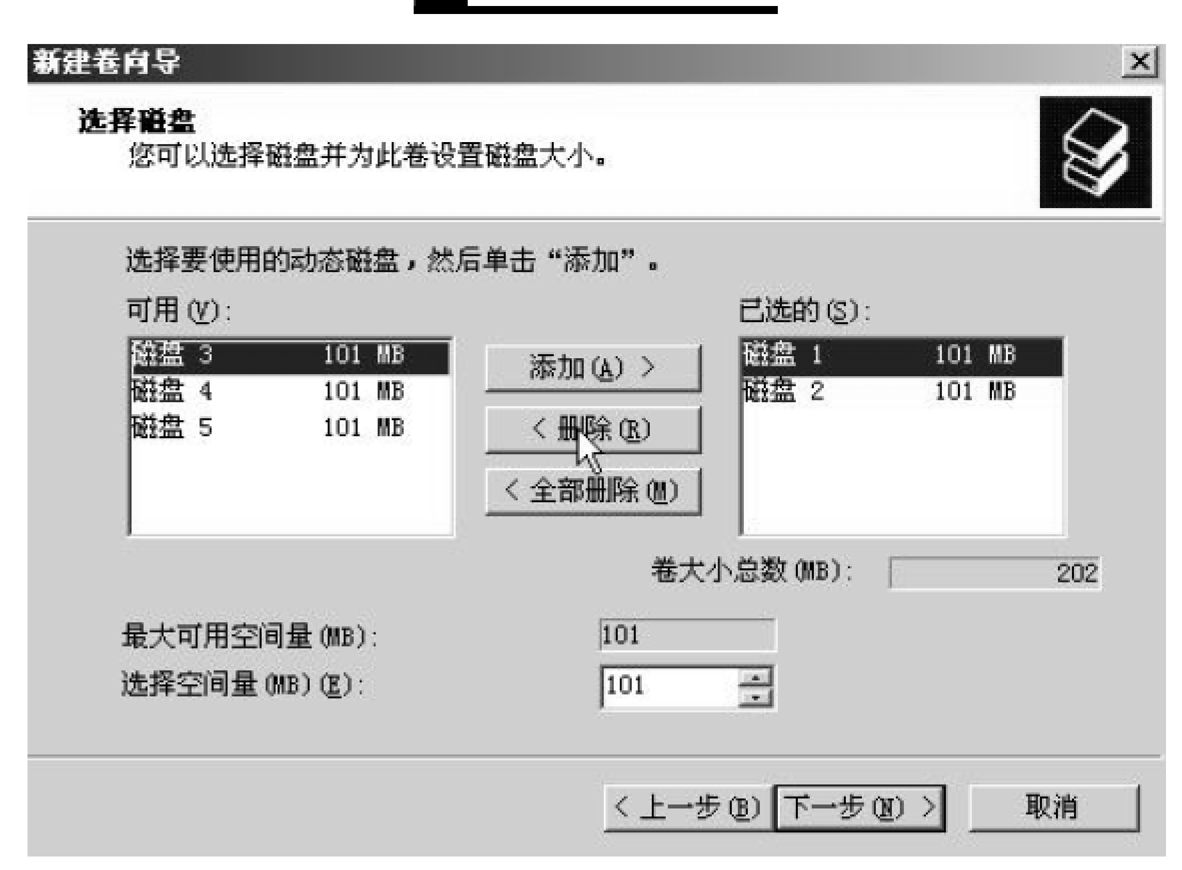


图5.9　跨区卷

2】 跨区卷允许卷容量来自多个硬盘，并且可以在每个硬盘上选择部分容量而不一定非要选择全部容量。在此，我们将全部容量划分给这个卷，卷总容量为 200MB，如图 5.10 所示。

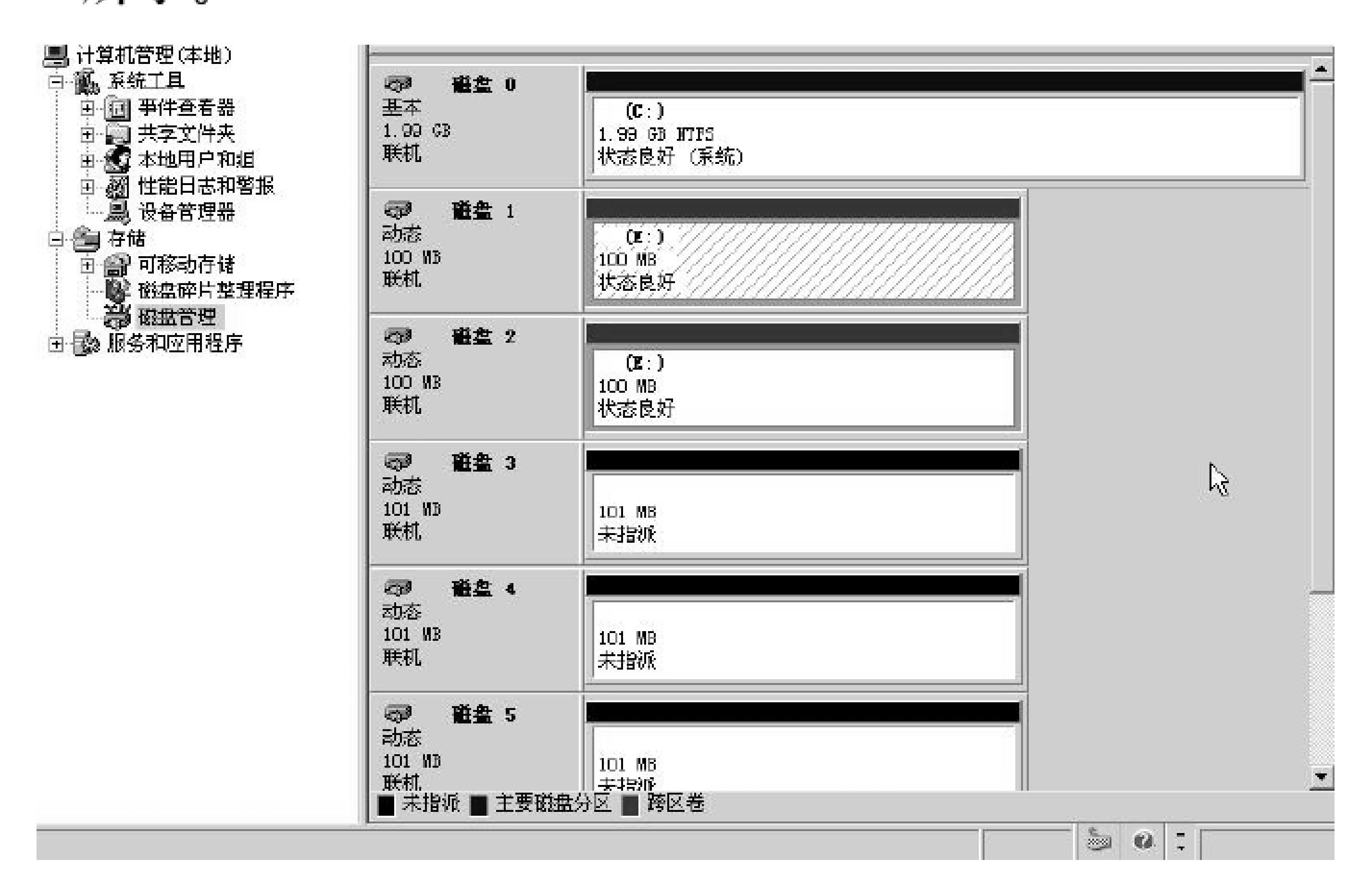


图5.10　跨区卷状态

3】 建好的跨区卷，将用紫色来表示。此外，还可以灵活地扩展这个卷的容量，如图 5.11 所示。

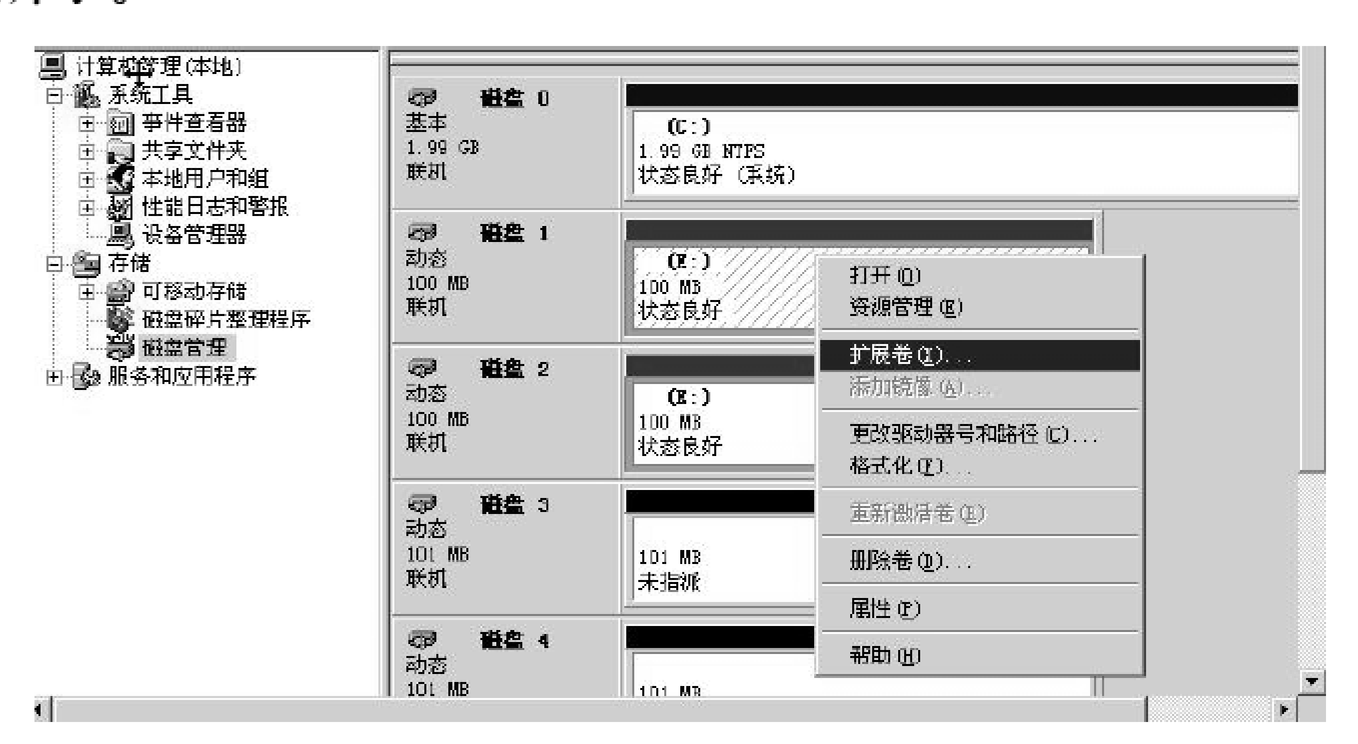


图 扩展容量

4】 我们向这个卷中再添加一块磁盘，磁盘 3，如图 5.12 所示。

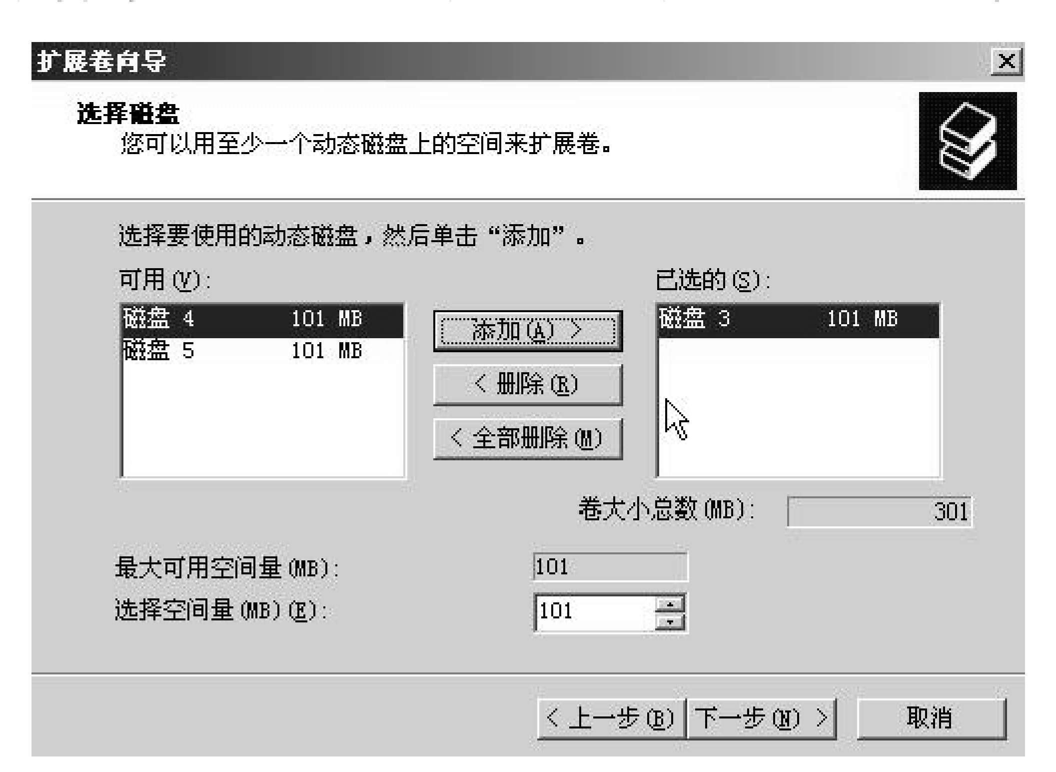


图 增加一块物理磁盘

5】 加完之后这个卷的容量就被扩充到了 300MB，如图 5.13 所示。

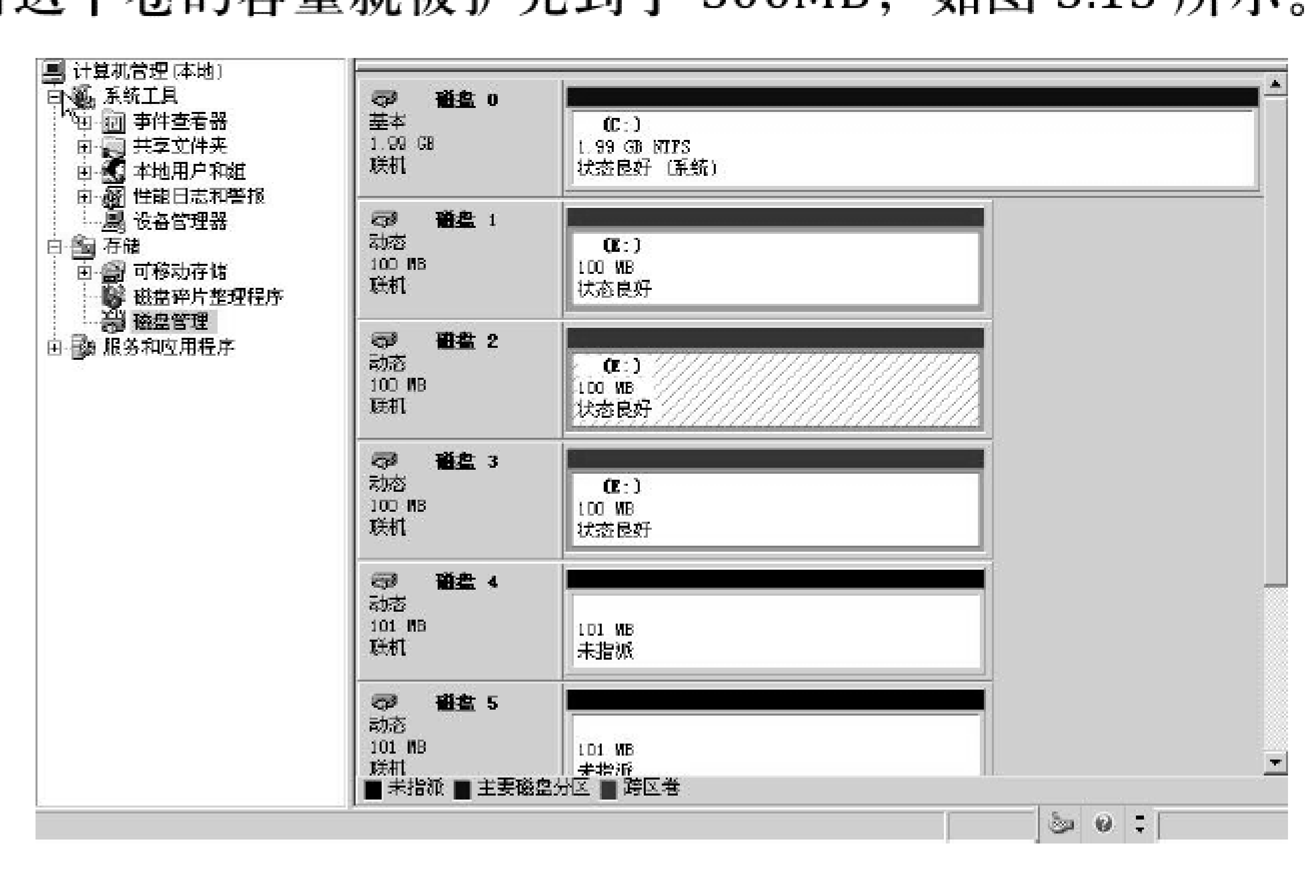


图 扩容后的卷

3. 删除卷

如 5.14 所示，可以 卷。

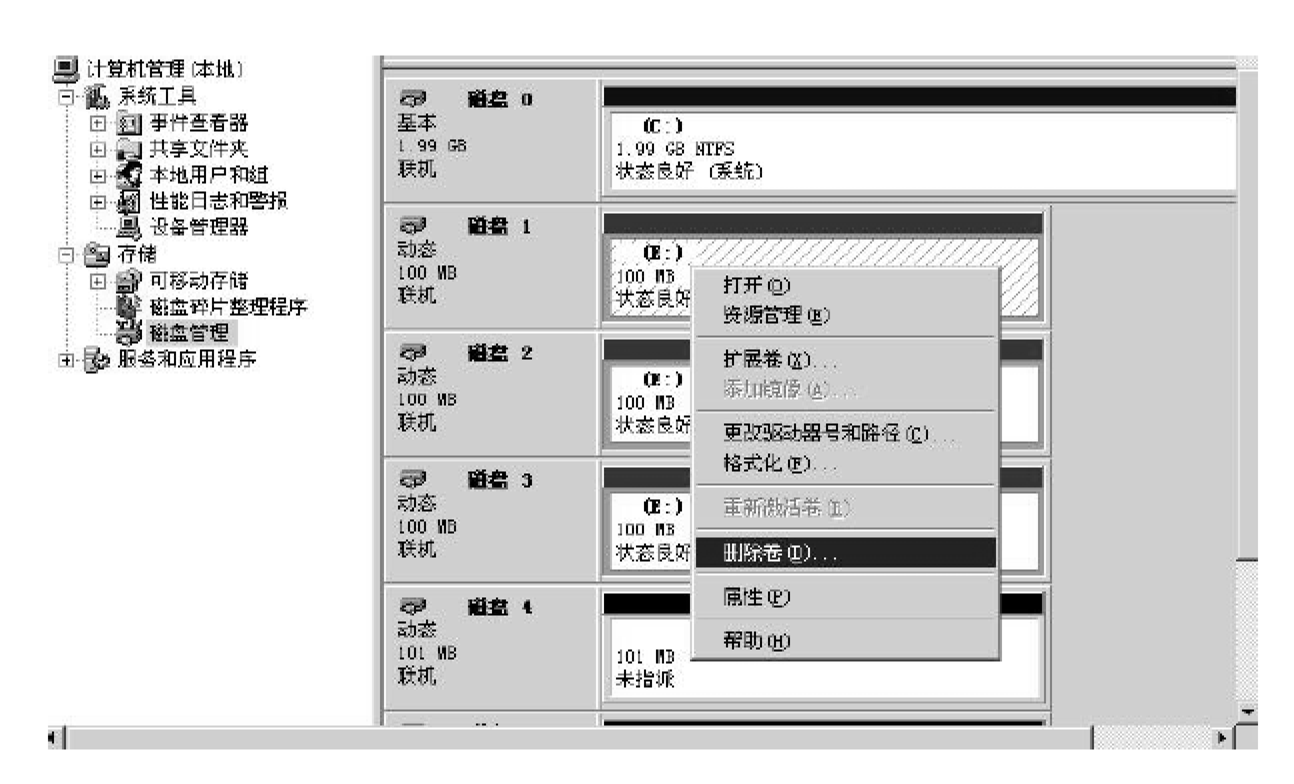


图5.14　删除卷

1】 下面用磁盘 1 的前 50MB 的容量和磁盘 2 的全部容量来做一个跨区卷，如图 5.15 所示。

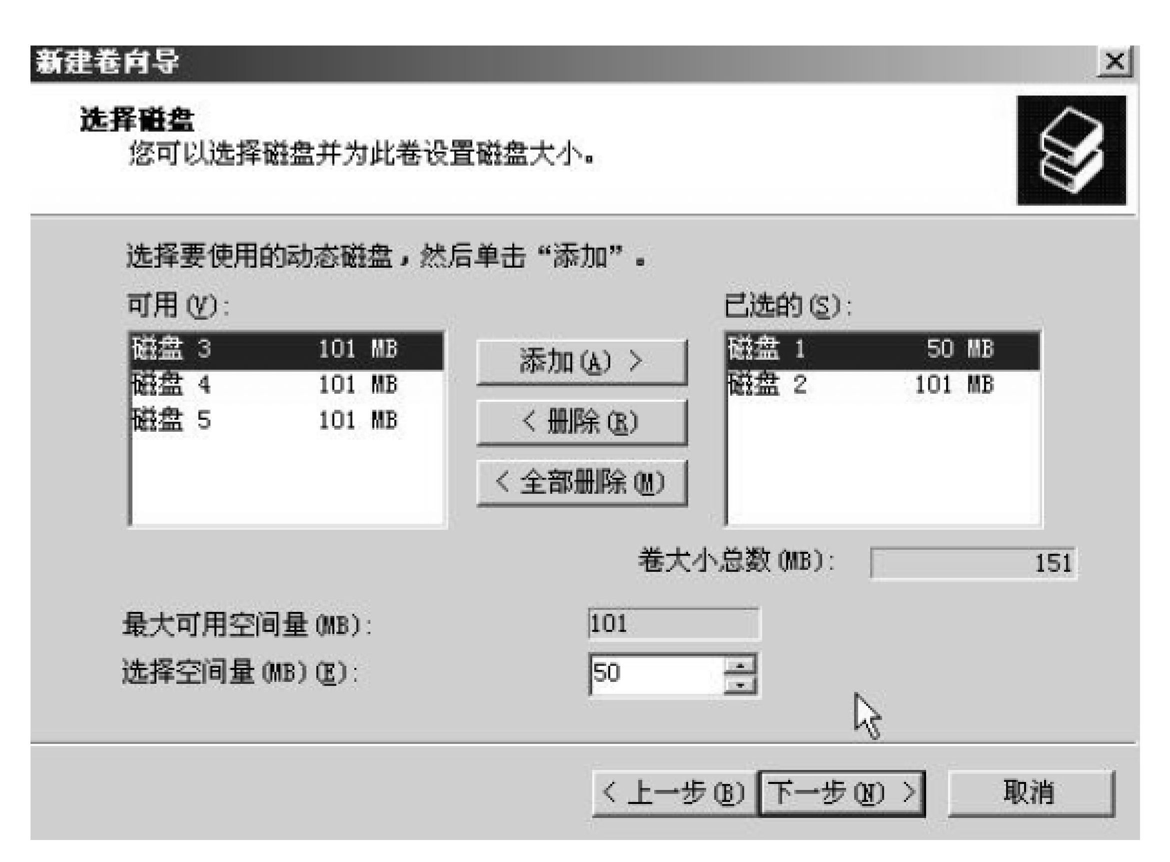


图5.15　灵活的划分尺寸

2】 做好后的卷如图 5.16 所示。此外，磁盘 1 剩余的 51MB 容量还可以再新建卷，如图 5.16 所示。

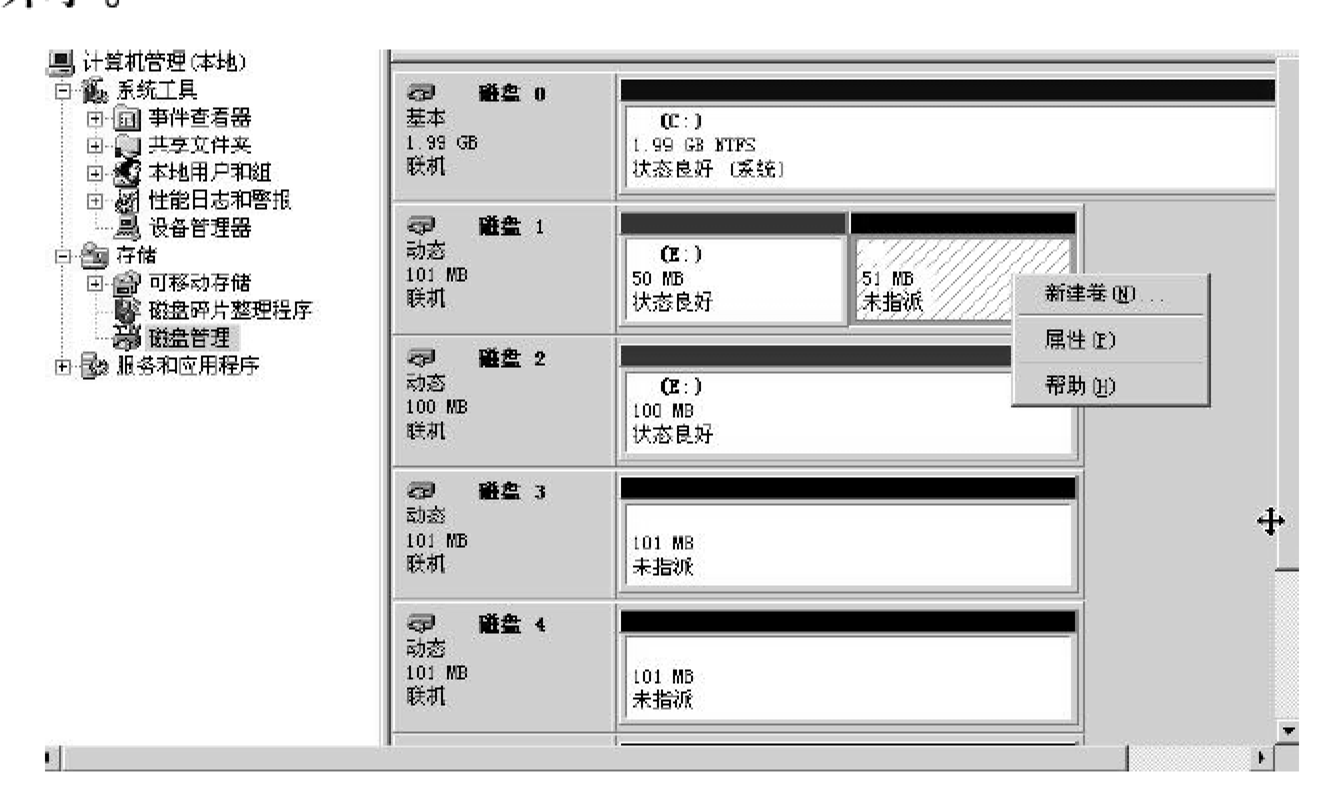


图5.16　剩余空间可以新建卷

4. 带区卷

下面我们来做一个带区卷，即条带化的 RAID 0 卷。我们选择用磁盘 1 和磁盘 2 中各

30MB 的容量来做一个 60MB 的卷，如图 5.17 和图 5.18 所示。

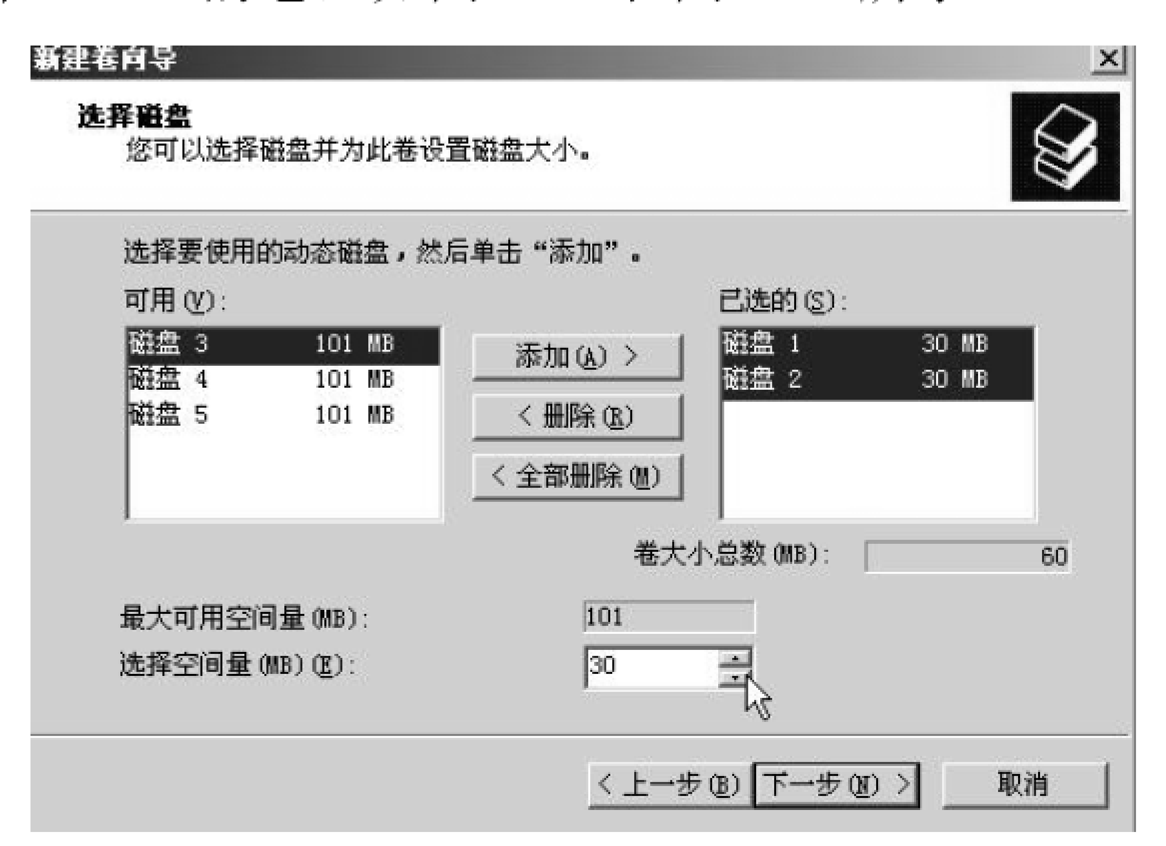


图5.17　带区卷

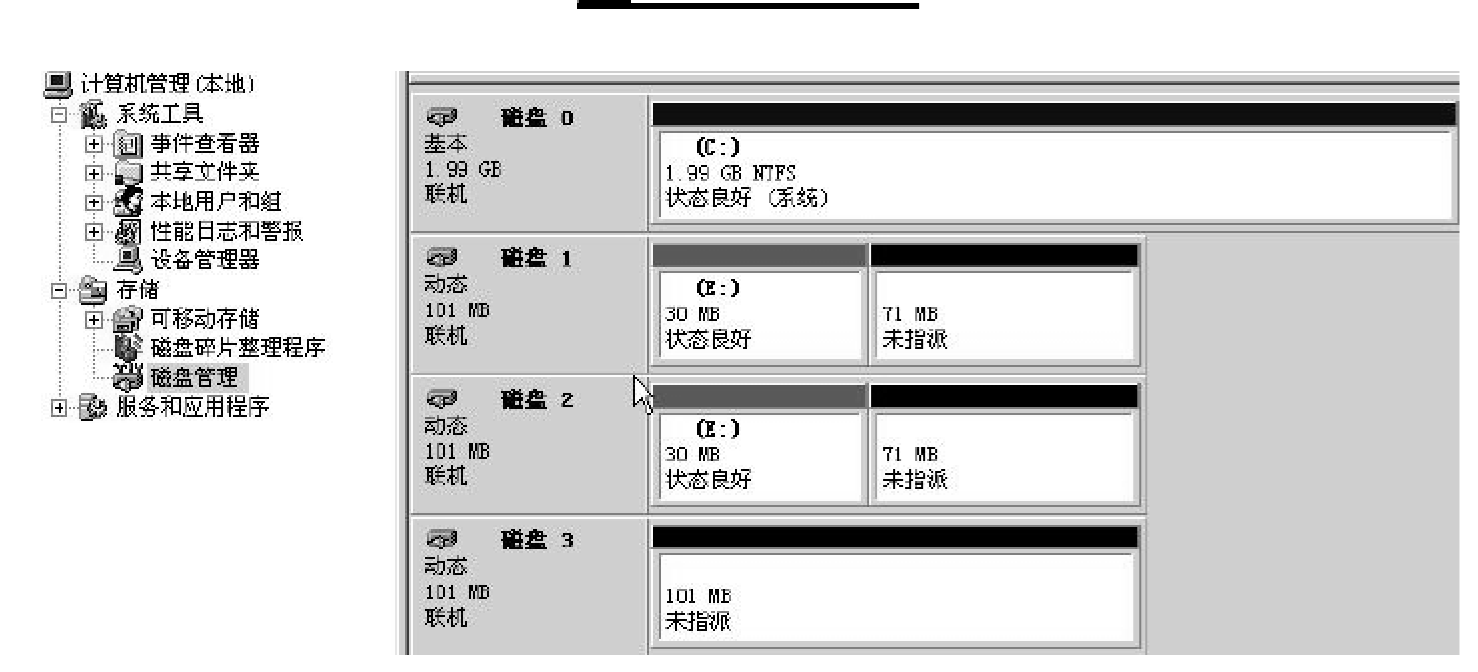


图5.18　带区卷的状态

做好之后的带区卷会用绿色标识。

5. 镜像卷

我们再来做一个镜像卷，即 RAID 1 卷。我们用磁盘 1 和磁盘 2 的各 40MB 容量来做一个 40MB 的卷，如图 5.19 和图 5.20 所示。

做好后的镜像卷会用棕色标识。

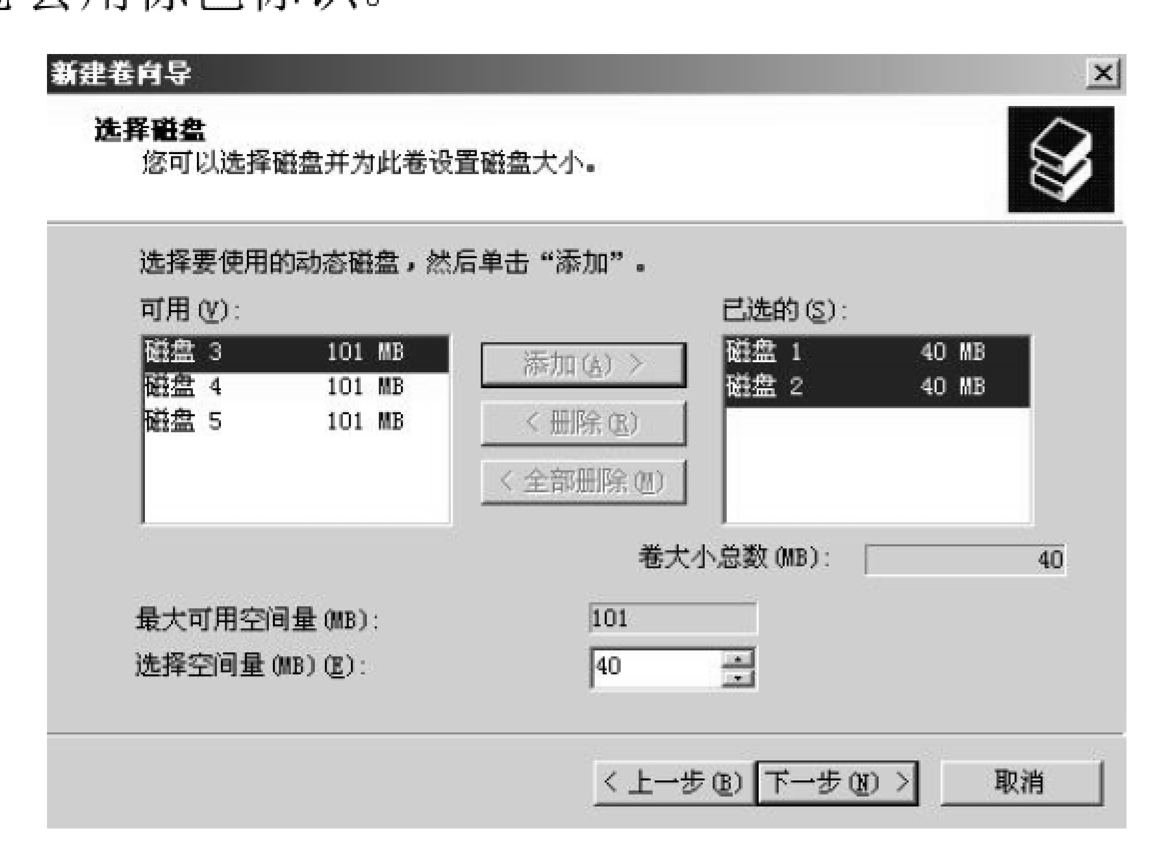


图5.19　镜像卷



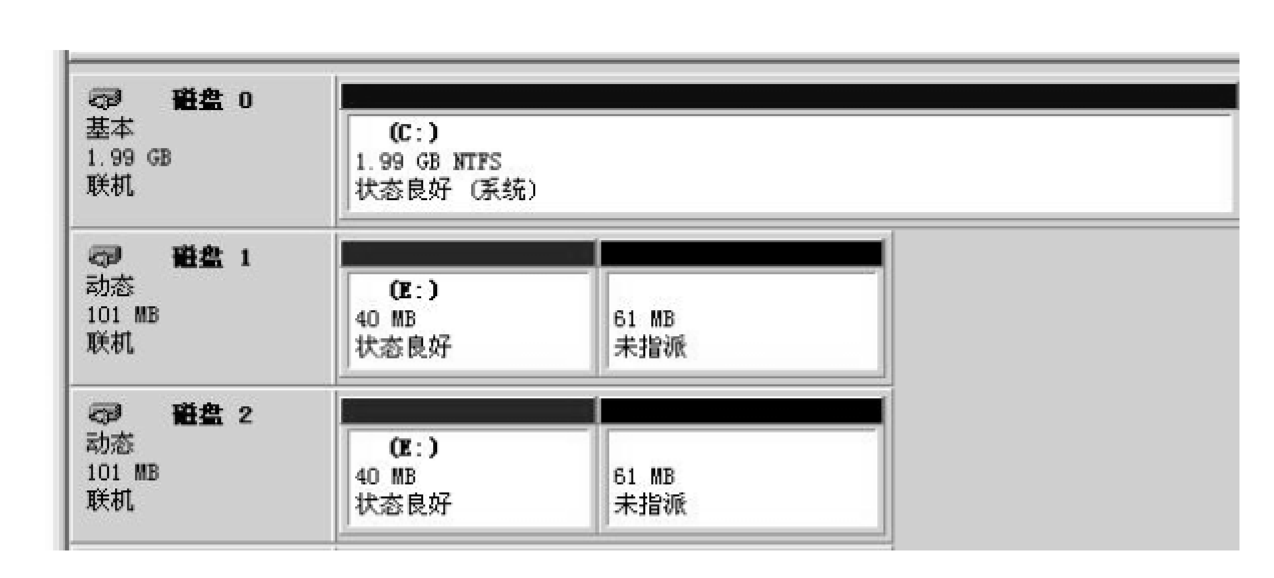


图5.20　镜像卷的状态

6. RAID 5 类型的卷

最后，我们来做一个 RAID 5 类型的卷，这里我们将所有磁盘的各 50MB 空间做一个卷，如图 5.21 所示。然后再用所有硬盘的 20MB 空间做一个卷，形成两个 RAID 5 卷。

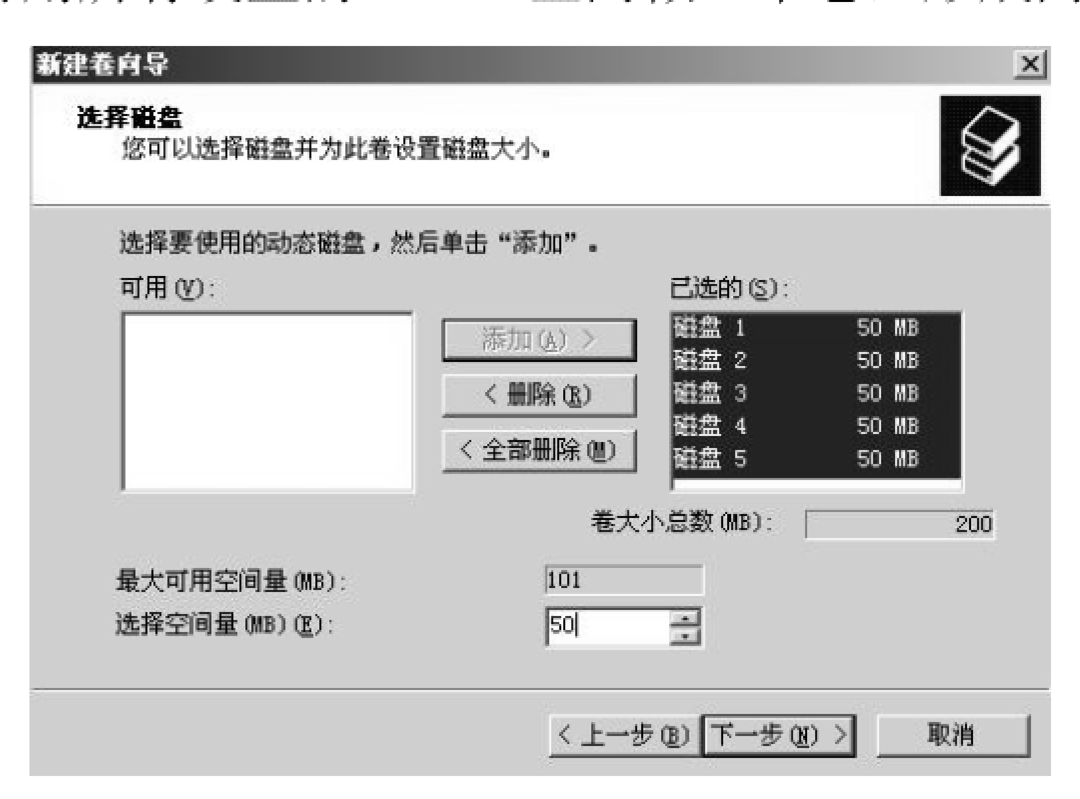


图5.21　创建 RAID 5 卷

做好后的 RAID 5 卷会用亮绿色标识，如图 5.22 所示。





图5.22　RAID 5 卷的状态



做好的任何卷均可随意被删除，如图 5.23 所示。



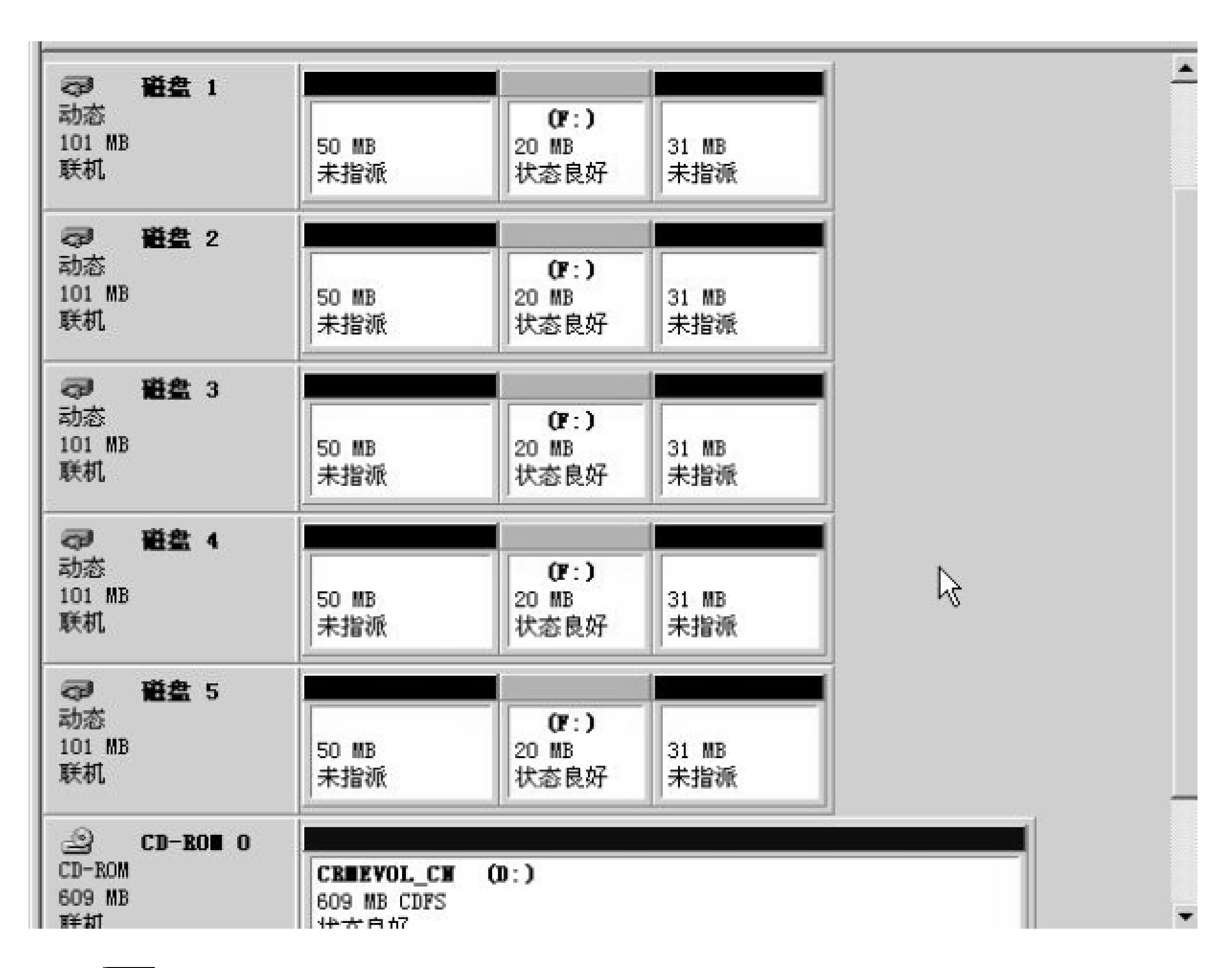


图5.23 删除了一个 RAID 5 卷

Windows 的动态磁盘管理实际上应该算是一个带有 RAID 功能的卷管理软件，而不仅仅是 RAID 软件。卷管理的概念我们在下文会解释。

5.1.2 Linux 下软 RAID 配置示例

我们将在一台装有 8 块物理磁盘的机器上安装 RedHat Enterprise Linux Server 4 Update 5 操作系统。具体操作过程如下

1】 选择手动配置磁盘界面，如图 5.24 所示。

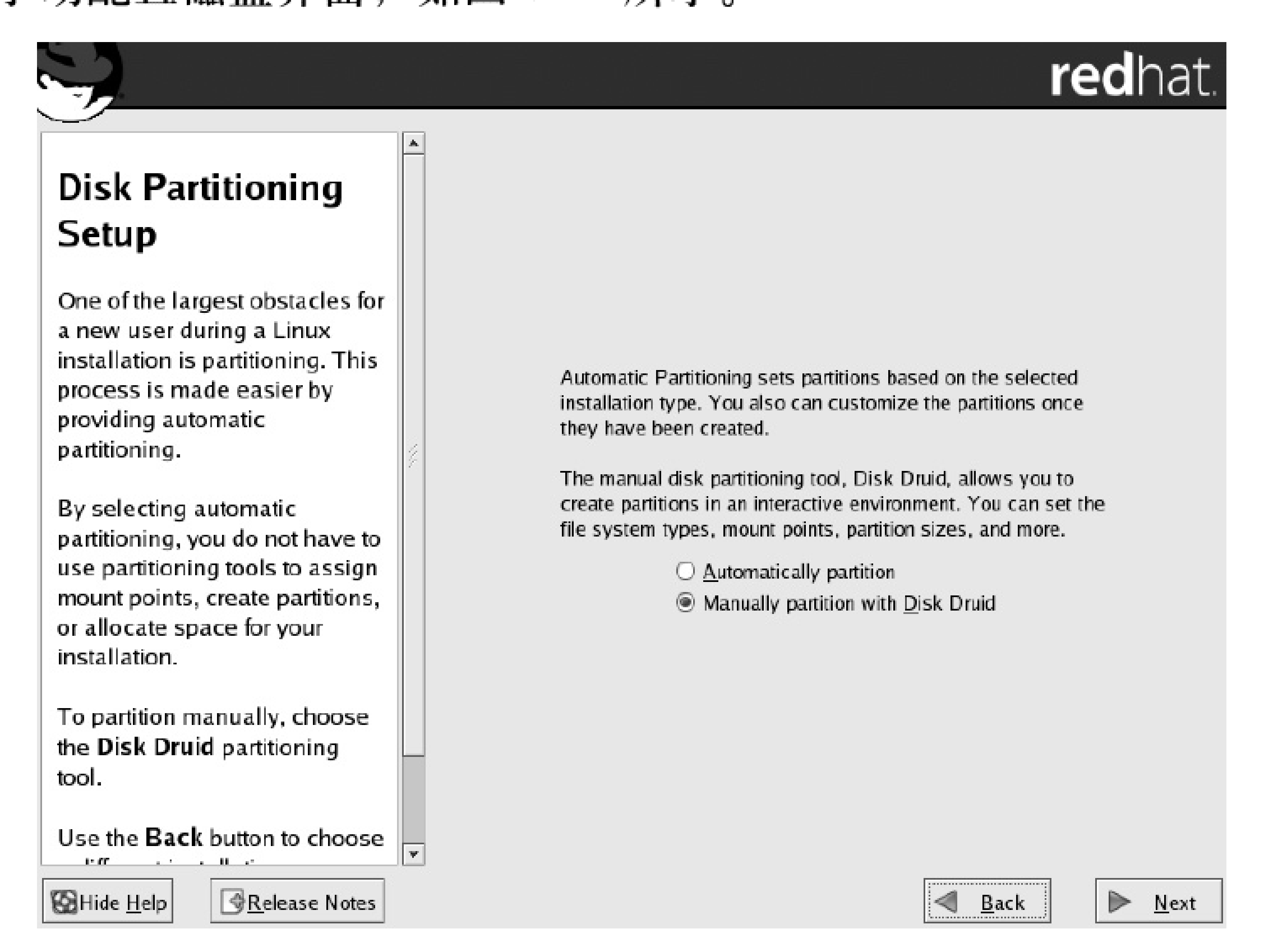


图5.24 选择手动配置

2】 可以看到系统识别到了 8 块物理磁盘，如图 5.25 所示。

3】 我们必须划分一个/boot 分区用来启动基本的操作系统内核。我们用第一块磁盘 sda 的前 100MB 容量来创建这个分区，如图 5.26 所示。

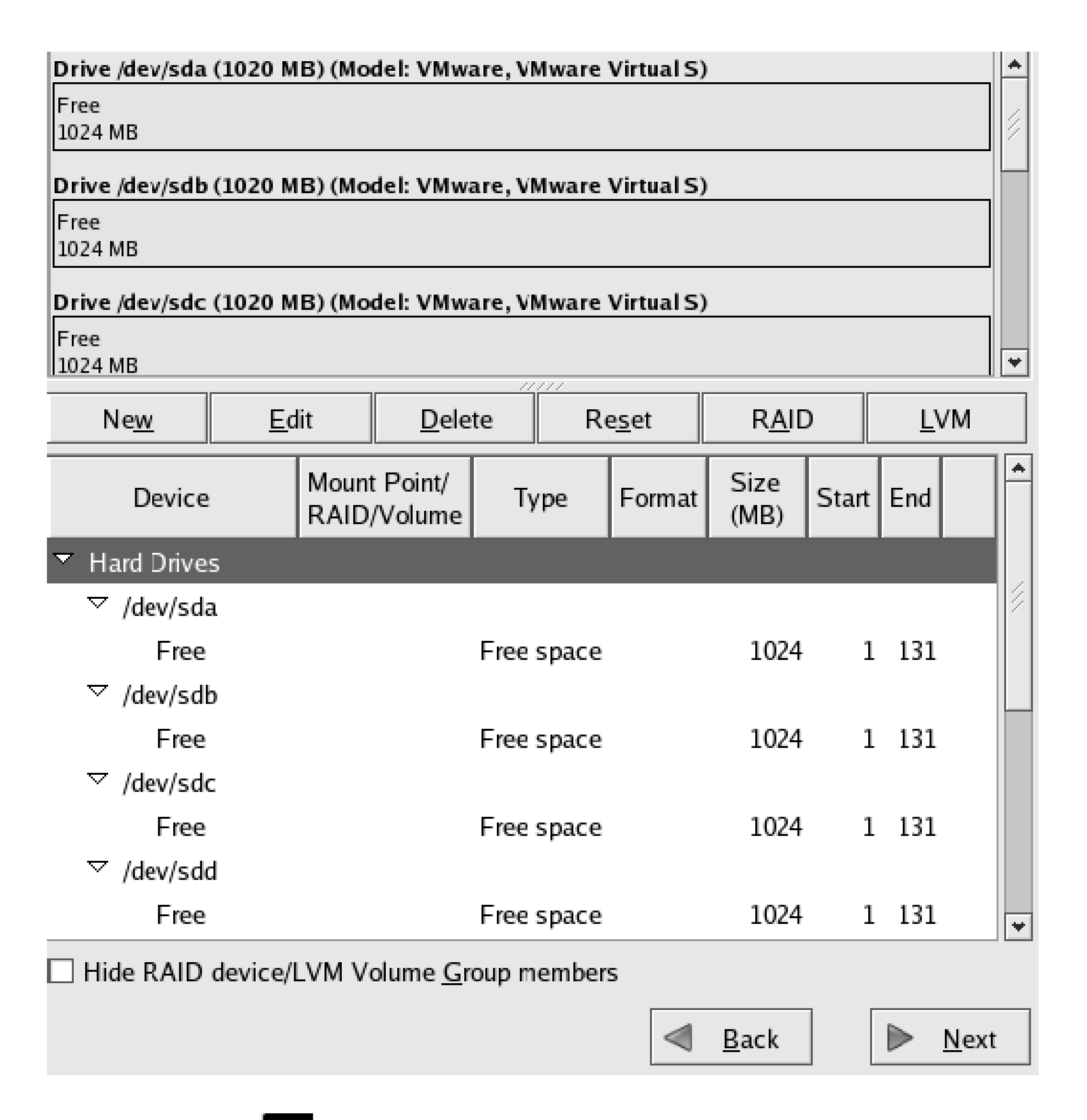


图5.25 识别到的磁盘列表

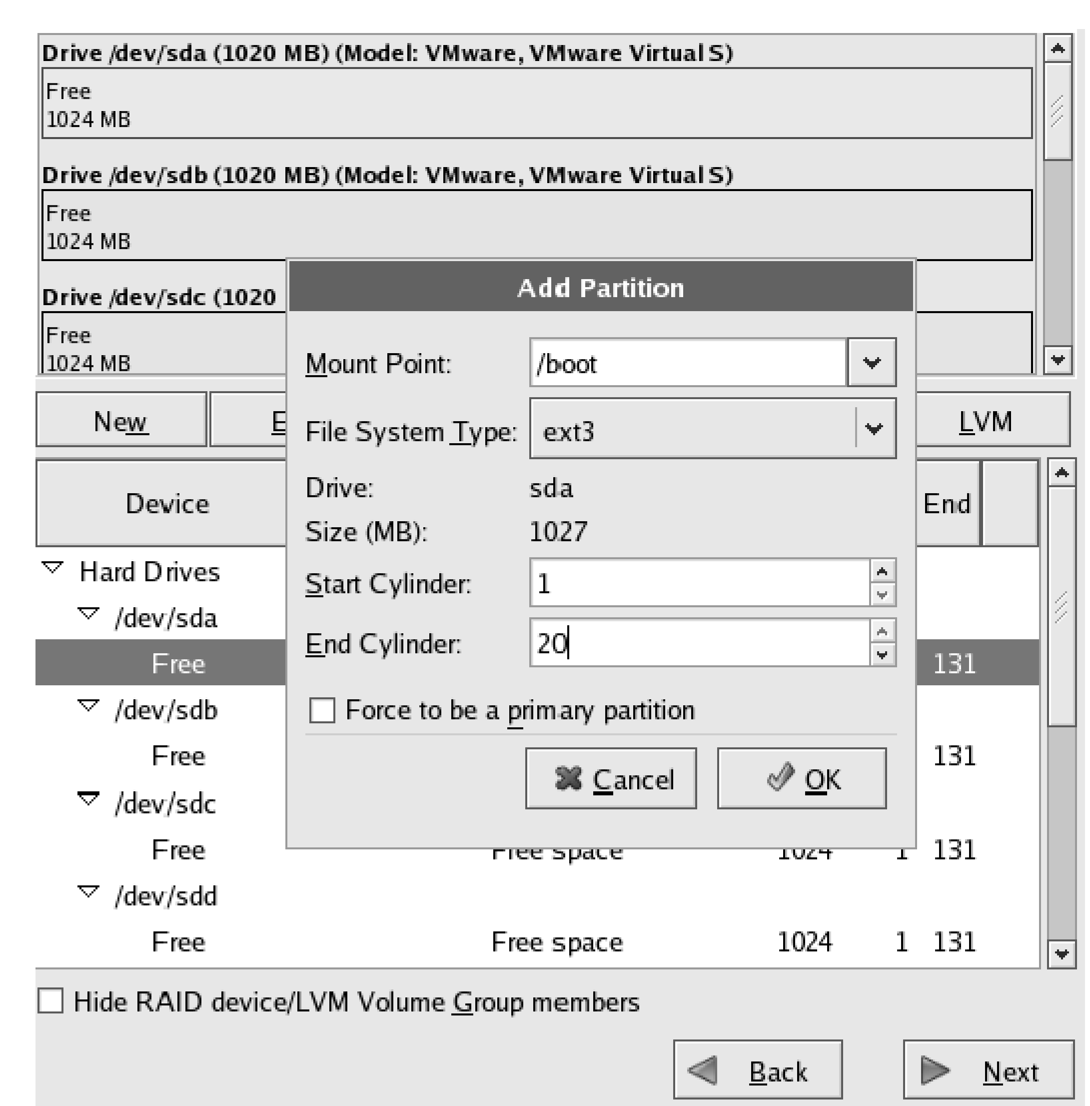


图5.26 创建/boot 分区

4】 在创建/boot 分区之后，将 sda 磁盘剩余的分区以及所有剩余的物理磁盘，均配置为 software RAID 类型，如图 5.27 所示。

5】 在将所有磁盘都配置成 software RAID 类型之后，单击 Next 按钮，会打开 RAID Options 对话框询问想要进行什么样的操作，如图 5.28 所示。

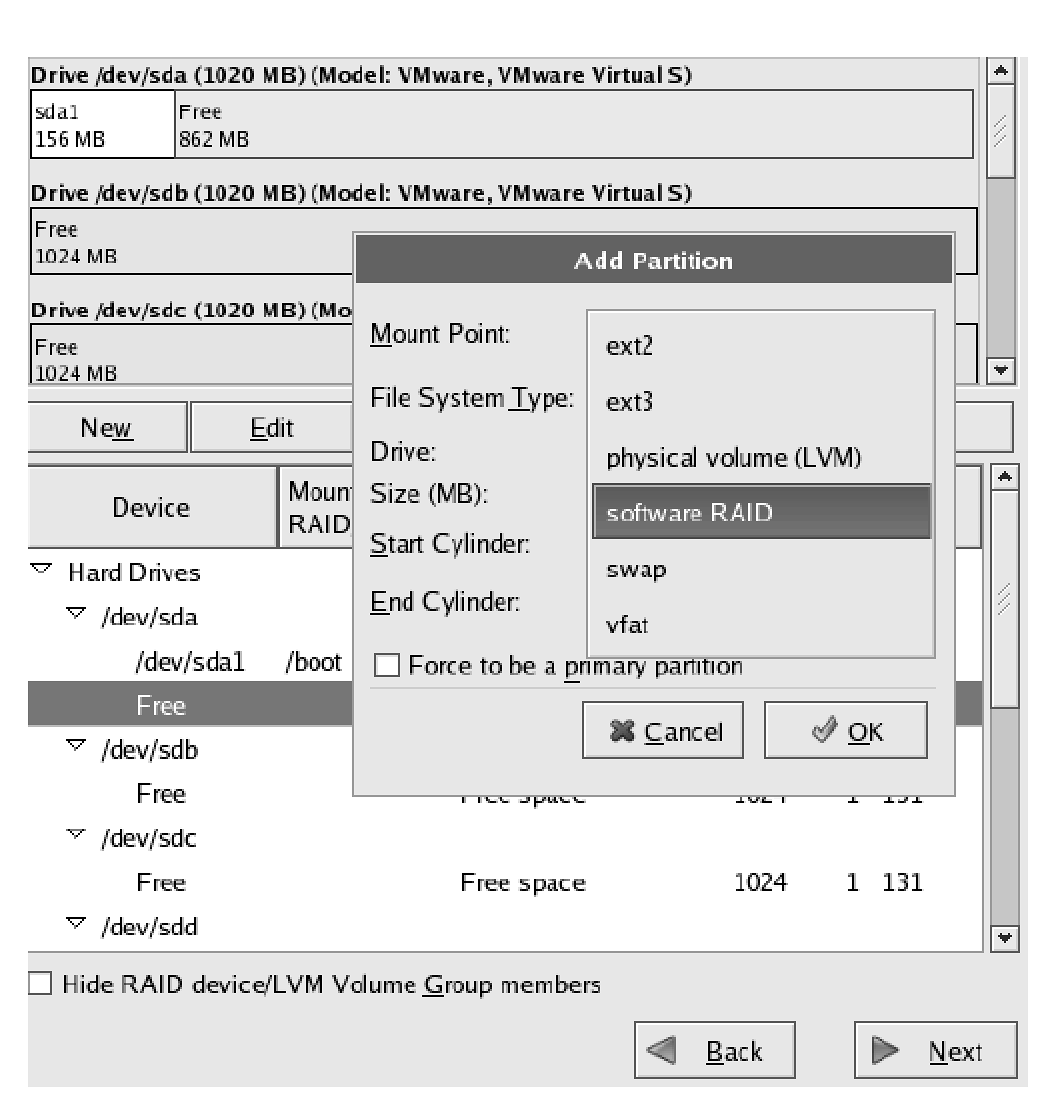


图5.27 配置磁盘类型

图5.28 设置为 RAID 设备

选中 Create a RAID device [default=/dev/md0]单选按钮后单击 OK 按钮，系统弹出 Make RAID Device 对话框。在对话框的 RAID Device 下拉列表框中，可以选择相应的 RAID 组在操作系统中对应的设备名。在 RAID Level 下拉列表框中，可以选择需要配置的 RAID

类型。在 RAID Members 列表框中，可以选择 RAID 组中包含的物理磁盘。用相同的方法可以做多个不同类型的 RAID 组，如图 5.29 所示。

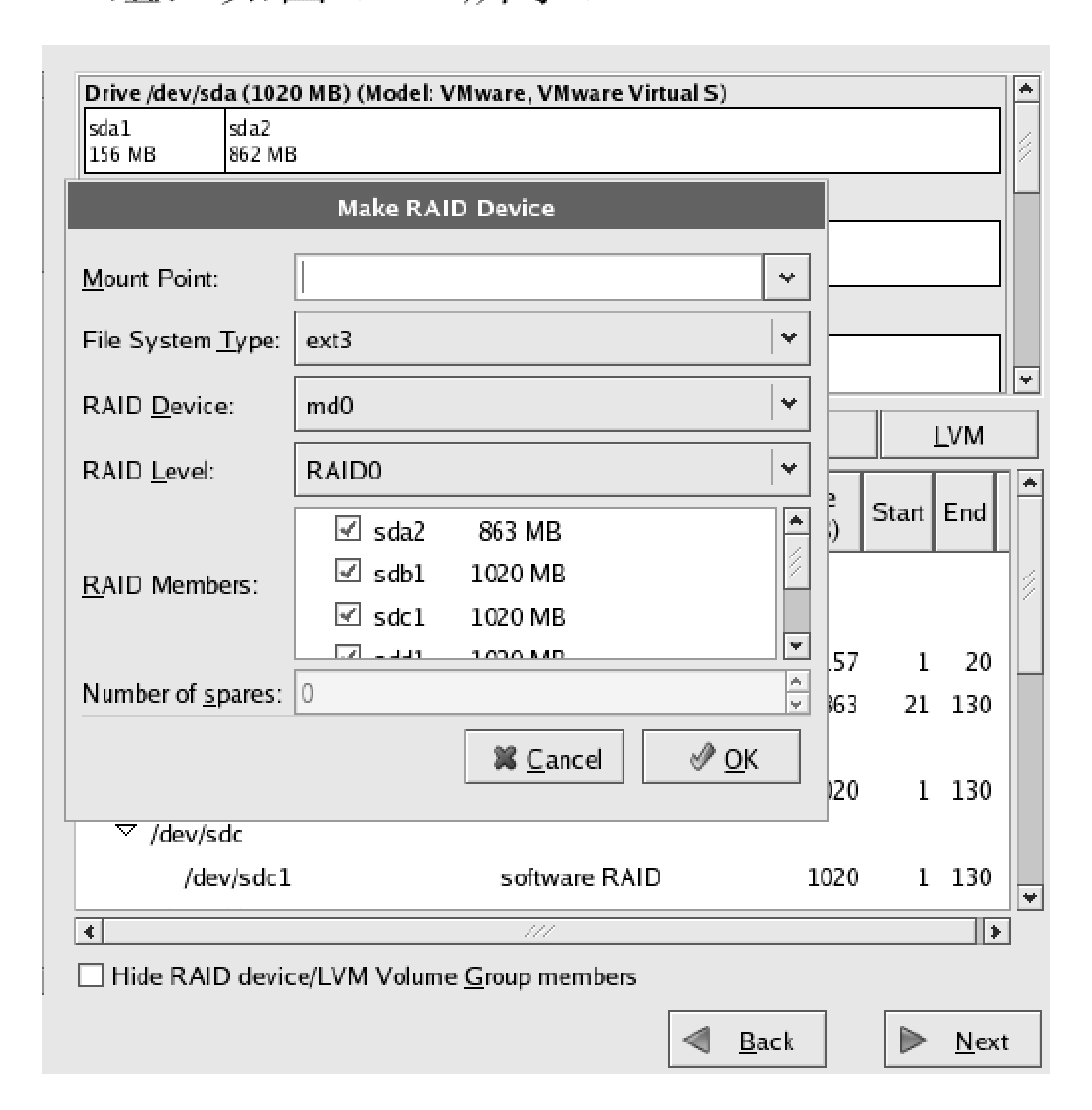


图5.29　创建对应的 Mount 点

5.2　RAID 卡

思考：软件 RAID 有三个缺点：第一，占用内存空间。第二，占用 CPU 资源。第三，软件 RAID 程序无法将安装有操作系统的那个磁盘分区做成 RAID 模式。因为 RAID 程序是运行在操作系统之上的，所以在启动操作系统之前，是无法实现 RAID 功能的。也就是说，如果操作系统损坏了，RAID 程序也就无法运行，磁盘上的数据就成了一堆无用的东西。因为 RAID 磁盘上的数据只有实现相应 RAID 算法的程序才能识别并且正确读写。如果没有相应的 RAID 程序，则物理磁盘上的数据仅仅是一些碎片而已，只有 RAID 程序才能组合这些碎片。幸好，目前大多数的 RAID 程序都会在磁盘上存储自己的算法信息，一旦操作系统出现了问题，或者主机硬件出现了问题，就可以将这些磁盘连接到其他机器上，再安装相同的 RAID 软件。RAID 软件读取了存储在硬盘上固定区域的 RAID 信息后，便可以继续使用。

软件 RAID 的缺点如此之多，使人们不断地思考更多实现 RAID 的方法。既然软件缺点太多，那么用硬件实现如何呢？

RAID 卡就是一种利用独立硬件来实现 RAID 功能的方法。要在硬件上实现 RAID 功能，必须找一个物理硬件作为载体，SCSI 卡或者主板上的南桥无疑就是这个载体了。人们在 SCSI 卡上增加了额外的芯片用于实现 RAID 功能。这些芯片是专门用来执行 RAID 算法的，可以是 ASIC 这样的高成本高速度运算芯片，也可以是通用指令 CPU 这样的通用代码执行芯片，可以从 ROM 中加载代码直接执行，也可以先载入 RAM 后执行，从而实现 RAID 功能。

实现了 RAID 功能的板卡(SCSI 卡或者 IDE 扩展卡)就叫做 RAID 卡。同样，在主板南桥芯片上也可实现 RAID 功能。由于南桥中的芯片不能靠 CPU 来完成它们的功能，所以这些芯片完全靠电路逻辑来自己运算，尽管速度很快，但是功能相对插卡式的 RAID 卡要弱。从某些主板的宣传广告中就可以看到，如所谓“板载”RAID 芯片就是指南桥中有实现 RAID 功能的芯片。

这样，操作系统不需要作任何改动，除了 RAID 卡驱动程序之外不用安装任何额外的软件，就可以直接识别到已经过 RAID 处理而生成的虚拟磁盘。

对于软件 RAID，至少操作系统最底层还是能感知到实际物理磁盘的，但是对于硬件 RAID 来说，操作系统根本无法感知底层的物理磁盘，而只能通过厂家提供的 RAID 卡的管理软件来查看卡上所连接的物理磁盘。而且，配置 RAID 卡的时候，也不能在操作系统下完成，而必须进入这个硬件来完成(或者在操作系统下通过 RAID 卡配置工具来设置)。一般的 RAID 卡都是在开机自检的时候，进入它的 ROM 配置程序来配置各种 RAID 功能。

RAID 卡克服了软件 RAID 的缺点，使操作系统本身可以安装在 RAID 虚拟磁盘之上，而这是软件 RAID 所做不到的。

1. RAID 卡的结构

带 CPU 的 RAID 卡俨然就是一个小的计算机系统，有自己的 CPU、内存、ROM、总线和 IO 接口，只不过这个小计算机是为大计算机服务的。

图 5.30 为一个 RAID 卡的架构示意图。

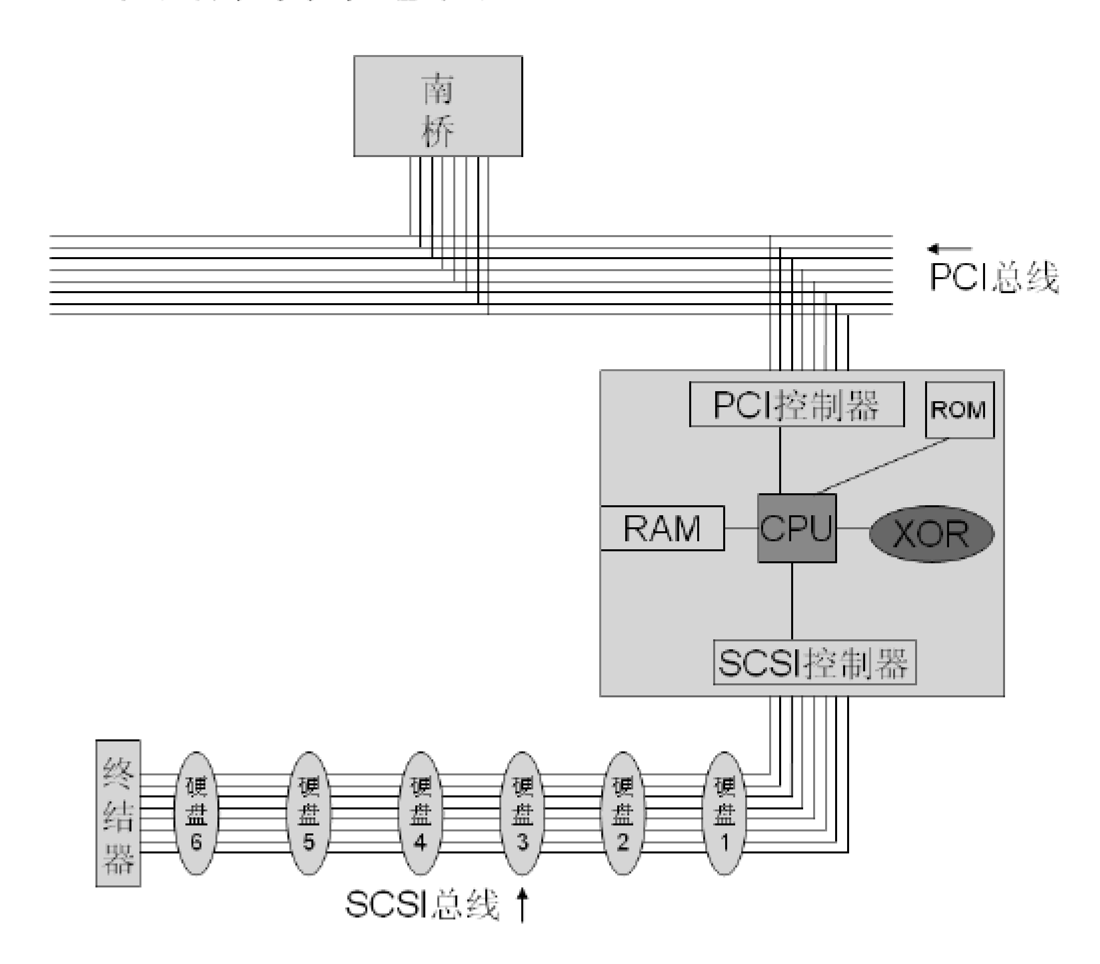


图5.30 RAID 卡结构示意图

SCSI RAID 卡上一定要包含 SCSI 控制器，因为其后端连接的依然是 SCSI 物理磁盘。其前端连接到主机的 PCI 总线上，所以一定要有一个 PCI 总线控制器来维护 PCI 总线的仲裁、数据发送接收等功能。还需要有一个 ROM，一般都是用 Flash 芯片作为 ROM，其中存放着初始化 RAID 卡必须的代码以及实现 RAID 功能所需的代码。

RAM 的作用，首先是作为数据缓存，提高性能；其次作为 RAID 卡上的 CPU 执行 RAID 运算所需要的内存空间。XOR 芯片是专门用来做 RAID 3、5、6 等这类校验型 RAID 的校验数据计算用的。如果让 CPU 来做校验计算，需要执行代码，将耗费很多周期。而如果直接使用专用的数字电路，一进一出就立即得到结果。所以为了解脱 CPU，增加了这块专门用

于 XOR 运算的电路模块，大大增加了数据校验计算的速度。

RAID 卡与 SCSI 卡的区别就在于 RAID 功能，其他没有太大区别。如果 RAID 卡上有多个 SCSI 通道，那么就称为多通道 RAID 卡。目前 SCSI RAID 卡最高有 4 通道的，其后端可以接入 4 条 SCSI 总线，所以最多可连接 64 个 SCSI 设备(16 位总线)。

增加了 RAID 功能之后，SCSI 控制器就成了 RAID 程序代码的傀儡，RAID 让它干什么，它就干什么。SCSI 控制器对它下面掌管的磁盘情况完全明了，它和 RAID 程序代码之间进行通信。RAID 程序代码知道 SCSI 控制器掌管的磁盘情况之后，就按照 ROM 中所设置的选项，比如 RAID 类型、条带大小等，对 RAID 程序代码做相应的调整，操控它的傀儡 SCSI 控制器向主机报告“虚拟”的逻辑盘，而不是所有物理磁盘了。



RAID 思想中有个条带化的概念。所谓的条带化，并不是真正的像低级格式化一样将磁盘划分成条和带。这个条带化完全就是在“心中”，也就是体现在程序代码上。因为条带的位置、大小一旦设置之后，就是固定的。一个虚拟盘上的某个 LBA 地址块，就对应了真正物理磁盘上的一个或者多个 LBA 块，这些映射关系都是预先通过配置界面设定好的。而且某种 RAID 算法往往体现为一些复杂公式，而不是去用一张表来记录每个虚拟磁盘 LBA 和物理磁盘 LBA 的对应，这样效率会很差。因为每个 IO 到来之后，RAID 都要查询这个表来获取对应物理磁盘的 LBA，而查询速度是非常慢的，更何况面对如此大的一张表。如果用一个逻辑 LBA 与物理 LBA 之间的函数关系公式来做运算，则速度是非常快的。

正是因为映射完全通过公式来进行，所以物理磁盘上根本不用写入什么标志，以标注所谓的条带。条带的概念只是逻辑上的，物理上并不存在。所以，条带等概念只需“记忆”在 RAID 程序代码之中就可以了，要改变也是改变程序代码即可。惟一要向磁盘上写入的就是一些 RAID 信息，这样即使将这些磁盘拿下来，放到同型号的另一块 RAID 卡上，也能无误地认出以前做好的 RAID 信息。SNIA 协会定义了一种 DDF RAID 信息标准格式，要求所有 RAID 卡厂家都按照这个标准来存放 RAID 信息，这样，所有 RAID 卡就都通用了。

条带化之后，RAID 程序代码就操控 SCSI 控制器向 OS 层驱动程序代码提交一个虚拟化之后的所谓“虚拟盘”或者“逻辑盘”，也有人干脆称为 LUN。

2. RAID 卡的初始化和配置过程

所谓初始化就是说在系统加电之后，CPU 执行系统总线特定地址上的第一句指令，这个地址便是主板 BIOS 芯片的地址。BIOS 芯片中包含着让 CPU 执行的第一条指令，CPU 将逐条执行这些指令，执行到一定阶段的时候，有一条指令会让 CPU 寻址总线上其他设备的 ROM 地址(如果有)。也就是说，系统加电之后，CPU 总会执行 SCSI 卡这个设备上 ROM 中的程序代码来初始化这块卡。初始化的内容包括检测卡型号、生产商以及扫描卡上的所有 SCSI 总线以找出每个设备并显示在显示器上。在初始化的过程中，可以像进入主板 BIOS 一样，进入 SCSI 卡自身的 BIOS 中进行设置，设置内容包括查看各个连接到 SCSI 总线上的设备的容量、生产商、状态、SCSI ID 和 LUN ID 等。

3. 0 通道 RAID 卡

0 通道 RAID 卡又称为 RAID 子卡，0 通道的意思是说这块卡的后端没有 SCSI 通道。将这块子卡插入主机的 PCI 插槽之后，它就可以利用主板上已经集成的或者已经插在 PCI 上的 SCSI 卡，来操控它们的通道，从而实现 RAID。这个 0 通道子卡，也是插到 PCI 上的一块卡，只不过它需要利用主板上为 0 通道子卡专门设计的逻辑电路，对外和 SCSI 控制器组成一块 RAID 卡来用，只不过这块卡在物理上被分割到了两个 PCI 插槽中而已。

图 5.31 展示了 0 通道 RAID 子卡的架构。在主板的一个特定 PCI 插槽上，有一个 ICR 逻辑电路，用来截获 CPU 发送的地址信号和发给 CPU 的中断信号。CPU 发送到这里原本用来操控 SCSI 控制器的地址信号，现在全部被这个 ICR 电路重定向到了 RAID 子卡处，包括主板 BIOS 初始载入 ROM，也不是载入 SCSI 卡的 ROM 了，而是载入了 RAID 子卡的 ROM。RAID 卡完全接替了 SCSI 卡来面对主机系统。RAID 卡和 SCSI 控制器的通信，包括地址信息和数据信息，需要占用 PCI 总线，这造成一定的性能损失。RAID 子卡和 SCSI 卡之间的通信，不会被 ICR 电路重定向。

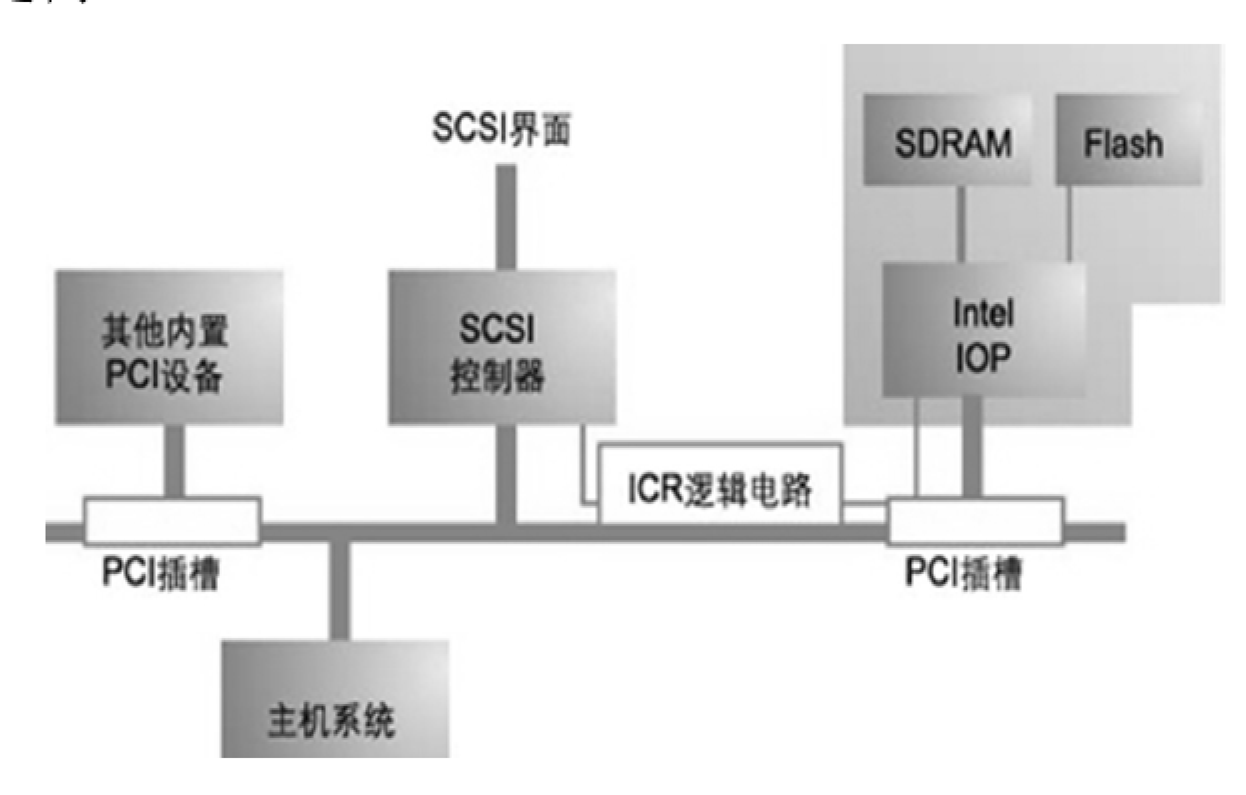


图5.31　0 通道 RAID 卡示意图

4. RAID On Chip 技术

ROC 技术是由 Adaptec 公司推出的一种廉价 RAID 技术，它利用 SCSI 卡上的 CPU 处理芯片，通过在 SCSI 卡的 ROM 中加入 RAID 代码而实现。

2001 年，Adaptec 展示了它的 iROC 技术，在 2003 年这一技术以 HOStRAID 的形象推出。iROC 也就是 RAID on CHIP，实质上就是利用 SCSI 控制芯片内部的 RISC 处理器完成一些简单的 RAID 类型(RAID 0、1、0+1)。由于 RAID 0、1 和 0+1 需要的运算量不大，利用 SCSI 控制器内部的 RISC 处理器也能够实现。在 ROM 代码的配合下，通过 iROC 实现的 RAID 0、1 或 0+1 具备引导能力，并且可以支持热备盘。

在入门级塔式服务器和 1U 高度的机架式服务器中，主板上通常会集成 SCSI 控制芯片，但不标配独立的 RAID 卡。iROC 的出发点就是让这些系统具有基本的硬件数据保护，当需要更为复杂的 RAID 5 时再购买独立的 RAID 卡。iROC 的出现给低端服务器产品的数据保护方案增加了一个简易的选择。iROC 或 HOStRAID 的主要缺点是操作系统兼容性和性能差，由于没有专门的 RAID 计算处理器，因此使用这种配置的 RAID 会在一定程度上降低服务器系统的性能，而且它只支持 RAID 0、1、0+1，只能支持几块 SCSI 盘做 RAID，相比 IDE RAID

0、1、0+1 来说特性相近而成本上却高了很多，此外，HOStRAID 技术在低端还必然要面对更新、性能更好的 S-ATA RAID 的竞争。

5. RAID 卡上的内存

RAID 卡上的内存，有数据缓存和代码执行内存两种作用。

RAID 卡上的 CPU 执行代码，当然需要 RAM 的参与了。如果直接从 ROM 中读取代码，速度会受到很大影响。所以 RAID 卡的 RAM 中有固定的地址段用于存放 CPU 执行的代码。而大部分空间都是用作了下文介绍的数据缓存。

缓存，也就是缓冲内存，只要在通信的双方之间能起到缓冲作用就可以了。我们知道 CPU 和内存之间是 L2 Cache，它比内存 RAM 速度还要高，但是没有 CPU 速度高。同样 RAID 控制器和磁盘通道控制器之间也要有一个缓存来适配，因为 RAID 控制器的处理速度远远快于通道控制器收集其通道上所连接的磁盘传出的数据速度。这个缓存没有必要用 L2 Cache 那样高速的电路，而用 RAM 足矣。因为 RAM 的速度就足够适配二者了。

缓存 RAM 除了适配不同速率的芯片通信之外，还有一个作用就是缓冲数据 IO。比如上层发起一个 IO 请求，RAID 控制器可以先将这个请求放到缓存中排队，然后一条一条地执行，或者优化这些 IO，能合并的合并，能并发的并发。

6. 缓存的两种写模式

对于上层的写 IO，RAID 控制器有两种手段来处理，内容如下。

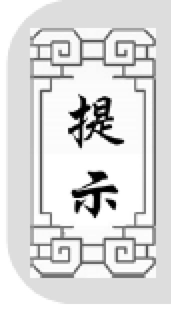
- WriteBack 模式：上层发过来的数据，RAID 控制器将其保存到缓存中之后，立即通知主机 IO 已经完成，从而主机可以不加等待地执行下一个 IO，而此时数据正在 RAID 卡的缓存中，而没有真正写入磁盘，起到了一个缓冲作用。RAID 控制器等待空闲时，或者一条一条地写入磁盘，或者批量写入磁盘，或者对这些 IO 进行排队(类似磁盘上的队列技术)等一些优化算法，以便高效写入磁盘。由于写盘速度比较慢，所以这种情况下 RAID 控制器欺骗了主机，但是获得了高速度，这就是“把简单留给上层，把麻烦留给自己”。这样做有一个致命缺点，就是一旦意外掉电，RAID 卡上缓存中的数据将全部丢失，而此时主机认为 IO 已经完成，这样上下层就产生了不一致，后果将非常严重。所以一些关键应用(比如数据库)都有自己的检测一致性的措施。也正因为如此，中高端的 RAID 卡都需要用电池来保护缓存，从而在意外掉电的情况下，电池可以持续对缓存进行供电，保证数据不丢失。再次加电的时候，RAID 卡会首先将缓存中的未完成的 IO 写入磁盘。
- WriteThrough 模式：Write Through 模式，也就是写透模式，即上层的 IO。只有切切实实被 RAID 控制器写入磁盘之后，才会通知主机 IO 完成，这样做保证了高可靠性。此时，缓存的提速作用就没有优势了，但是其缓冲作用依然有效。

除了作为写缓存之外，读缓存也是非常重要的。缓存算法是门很复杂的学问，有一套复杂的机制，其中一种算法叫做 PreFetch，即预取，也就是对磁盘上接下来“有可能”被主机访问到的数据，在主机还没有发出读 IO 请求的时候，就“擅自”先读入到缓存。这个“有可能”是怎么来算的呢？

其实就是认为主机下一次 IO，有很大几率会读取到这一次所读取的数据所在磁盘位置相邻位置的数据。这个假设，对于连续 IO 顺序读取情况非常适用，比如读取逻辑上连续存

放的数据，这种应用如 FTP 大文件传输服务、视频点播服务等，都是读大文件的应用。而如果很多碎小文件也是被连续存放在磁盘上相邻位置的，缓存会大大提升性能，因为读取小文件需要的 IOPS 很高，如果没有缓存，全靠磁头寻道来完成每次 IO，耗费时间是比较长的。

还有一种缓存算法，它的思想不是预取了，它是假设：主机下一次 IO，可能还会读取上一次或者上几次(最近)读取过的数据。这种假设和预取完全不一样了，RAID 控制器读取出一段数据到缓存之后，如果这些数据被主机的写 IO 更改了，控制器不会立即将它们写入磁盘保存，而是继续留在缓存中，因为它假设主机最近可能还要读取这些数据，既然假设这样，那么就没有必要写入磁盘并删除缓存，然后等主机读取的时候，再从磁盘读出来到缓存，还不如以静制动，干脆就留在缓存中，等主机“折腾”的频率不高了，再写入磁盘。



中高端的 RAID 卡一般具有 256MB 的 RAM 作为缓存。

7. RAID 配置完后的初始化过程

对于校验型 RAID，在 RAID 卡上设置完 RAID 参数并且应用 RAID 设置之后，RAID 阵列中的所有磁盘需要进行一个初始化过程，所需要的时间与磁盘数量、大小有关。磁盘越大，数量越多，需要的时间就越长。



RAID 卡都向磁盘上写了什么东西呢？大家可以想一下，一块刚刚出厂的新磁盘，上面有没有数据？

有。具体什么数据呢？要么全是 0，要么全是 1。这里所说的全 0 是指实际数据部分，扇区头标等一些特殊位置除外。因为磁盘上的磁性区域就有两种状态，不是 N 极，就是 S 极。那么也就是说不是 0 就是 1，而不可能有第三种状态。那么这些 0 或者 1，算不算数据呢？当然要算了。如果此时用几块磁盘做了 RAID 5，但磁盘上任何数据都不做改动，我们看一下此时会处于一种什么状态，比如 5 块磁盘，4 块数据盘空间，1 块校验盘空间，同一个条带上，4 块数据块，一块校验块，所有块上的数据都是全 0，那么此时如果按照 RAID 5 来算，是正确的，因为 0 xor 0 xor 0 xor 0 xor 0 = 0，对。

如果一开始磁盘全是 1，那么同样地 1 xor 1 xor 1 xor 1 xor 1 = 1，也对。但是如果用 6 块盘做 RAID 5，而且初始全为 1，情况就矛盾了。1 xor 1 xor 1 xor 1 xor 1 xor 1 = 0，此时正确结果应该是校验块为 0，但是初始磁盘全部为 1，校验块的数据也为 1，这就和计算结果相矛盾了。

如果初始化过程不对磁盘数据进行任何更改，直接拿来写数据，比如此时就向第二个 extend 上写了一块数据，将 1 变为 0，然后控制器根据公式：新数据的校验数据=(老数据 EOR 新数据) EOR 来校验数据。(1 eor 0) eor 1 = 0，新校验数据为 0，所以最终数据变成了这样：1 xor 0 xor 1 xor 1 xor 1 xor 1。我们算出它的正确数据应该等于 1，而由 RAID 控制器算的却成了 0，所以就矛盾了。

为什么会犯这个错误呢？那是因为一开始 RAID 控制器就没有从一个正确的数据关系

开始算，校验块的校验数据一开始就与数据块不一致，导致越算越错。所以 RAID 控制器在做完设置，并启用之后，在初始化的过程中需要将磁盘每个扇区都写成 0 或者 1，然后计算出正确的校验位，或者不更改数据块的数据，直接用这些已经存在的数据，重新计算所有条带的校验块数据。在这个基础上，新到来的数据才不会被以讹传讹。



像 NetApp 等产品，其 RAID 组做好之后不需要初始化，立即可用。甚至向已经有数据的 RAID 组中添加磁盘，也不会造成任何额外的 IO。因为其会将所有 Spare 磁盘清零，也就是向磁盘发送一个 Zero Unit 的 SCSI 指令，磁盘会自动执行清零。用这些磁盘做的 RAID 组，不需要校验纠正，所以也不需要初始化过程，或者说初始化过程就是等待磁盘清零的过程。

8. 几款 RAID 卡介绍

1) Mylex AcceleRAID 352

双通道 160M 部门级，性能强悍， BIOS 选项极为人性化，在 BIOS 内可以检测 SCSI 硬盘的出厂坏道及成长坏道，而不需借助软件。并且允许手动打开/关闭硬盘设备自身的 Read cache/Write cache。还带有电池。

详细信息如下。

- 支持 RAID 级别：RAID 0、1、0+1、3、5、10、30、50、JBOD。
- 处理芯片：Intel i960RN。
- 总线类型：PCI 64bit 兼容 32bit。
- 外置接口：Ultra 160 SCSI。
- 数据传输速率：最高 160MB/s。
- 外接设备数：最多 30 个 SCSI 外设。
- 内部接口：双 68 针高密。
- 外部接口：双 68 针超高密。
- 适用的操作系统：Windows NT 4.0；Windows 2K；NetWare 4.2、5.1；SCO OpenServer 5.05、5.0.6；SCO UnixWare 7.1；DOS 6.x and above；Solaris 7 (x86)；Linux 2.2 kernel distributions。
- 包括软件：有 Storager Manager，Storager Manager Pro 和 CLI(命令分界面)。
- 主要 RAID 特性：在线扩容、瞬时阵列可用性(后台初始化)、支持 S.M.A.R.T、支持 SES/SAF-TE。

图 5.32 和图 5.33 为 Mylex AcceleRAID 352 卡实物图。

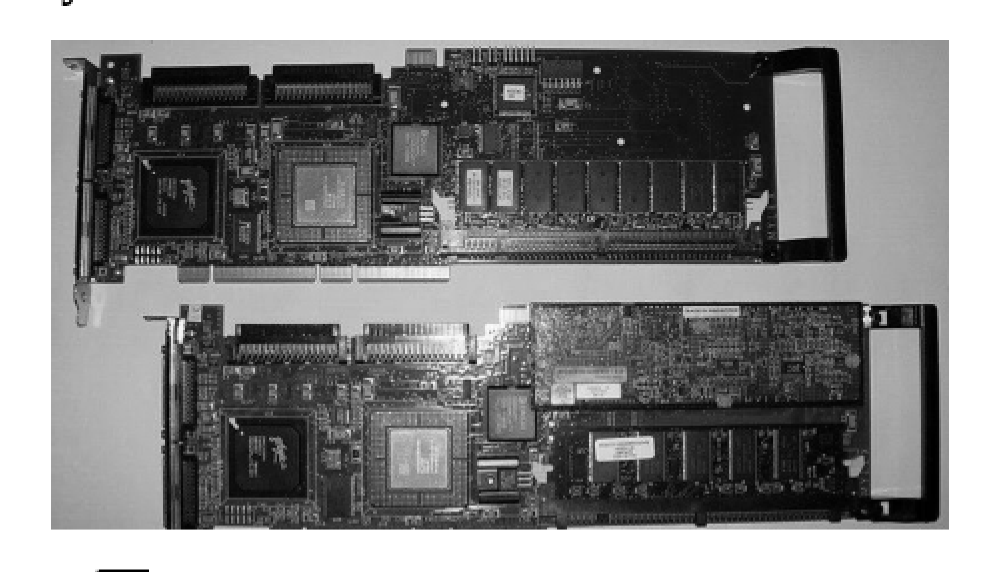


图 5.32 Mylex AcceleRAID 352 卡(1)



图5.33 Mylex AcceleRAID 352 卡(2)

2) LSI MegaRAID Enterprise 1600(AMI 471)

4 通道 160M 企业/部门级，160M 最为顶级豪华的 SCSI RAID，强大的 BIOS 选项(LSI 独有的 Web BIOS)。卡上系统缓存可详细调节(除大部分 SCSI RAID 可以调节的主要功能 Write back(回写)外，增加 Read ahead(预读)，Cache I/O 等可调选项，满足 RAID 的用途需要，体现各种 RAID 的最高性能，带电池。

详细信息如下。

- 支持 RAID 级别：RAID 0、1、0+1、3、5、10、30、50、JBOD。
- 处理芯片：Intel i960RN。
- 插槽类型：PCI 64bit、兼容 32bit。
- 总线速度：66MHz。
- 总线宽度：64bit。
- 外置接口：Ultra 160 SCSI。
- 数据传输率：160MB/s。
- 最多连接设备：32。
- 内部接口：双 68 针高密。
- 外部接口：四 68 针超高密。
- 系统平台：Windows95/98/Me/4.0/2000/XP，Linux(Red Hat、SuSE、Turbo、Caldera 和 FreeBSD)。

图 5.34 为 LSI MegaRAID Enterprise 1600 卡实物图。

可以看到 RAID 卡使用的内存就是台式机的 SDRAM 内存，有些使用 DDR SDRAM 内存。

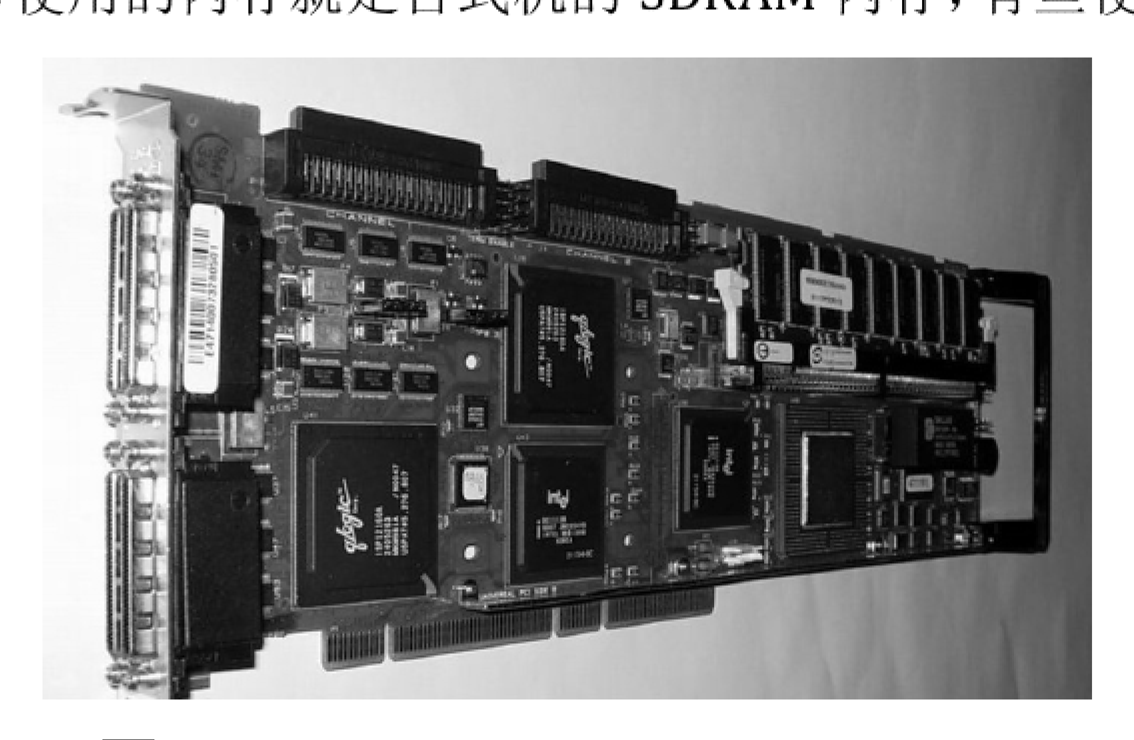


图5.34 LSI MegaRAID Enterprise 1600 卡

9. 用 Rocket RAID 卡做各种 RAID

在一张 Rocket RAID 卡上，安装了 8 块 IDE 磁盘。开机之后，在启动界面按照相应提示进入 RAID 卡的设置界面，如图 5.35 所示。

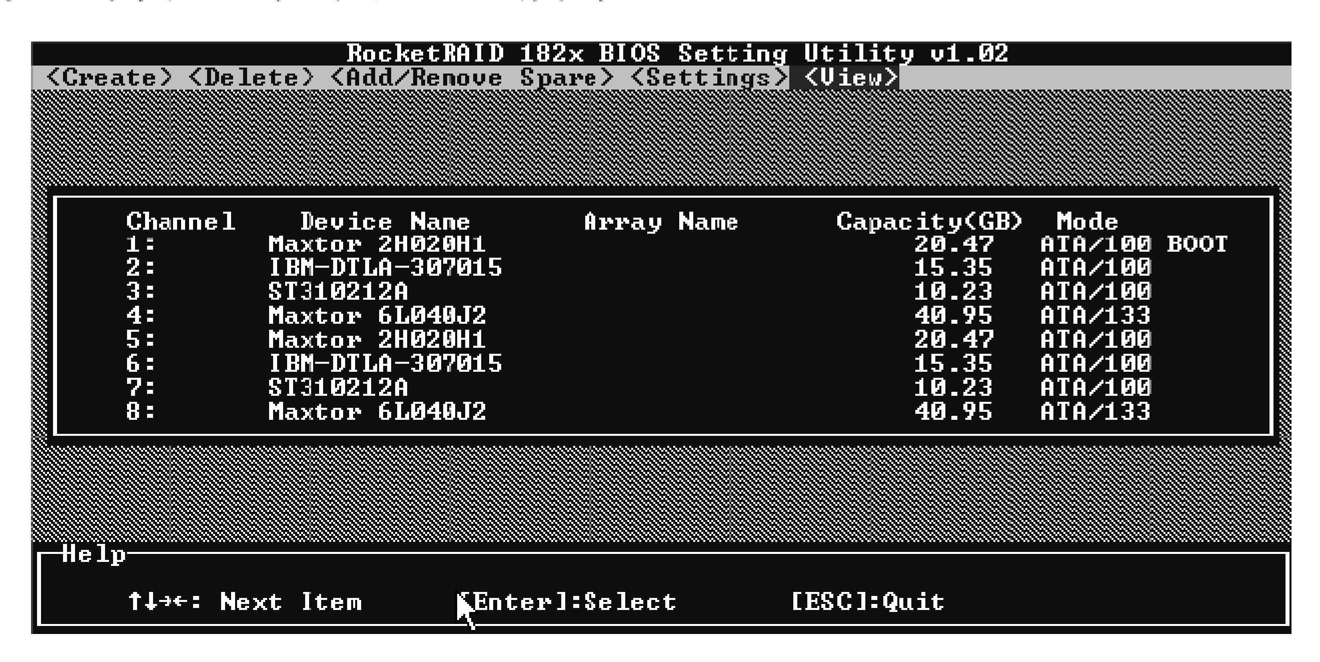


图5.35　磁盘列表

可以看到，这 8 块硬盘有着不同的品牌、容量以及参数，但它们都是 IDE 接口的 ATA 硬盘。

1)　RAID 0 组创建过程

RAID 0 组创建过程如下

1】 选择 RAID 0：Striping，如图 5.36 所示。

2】 给新 RAID 0 组起名为“RAID 0”，如图 5.37 所示。

3】 在 Select Devices 菜单下，选择 RAID 0 组所包含的磁盘，如图 5.38 和图 5.39 所示。

4】 接下来，在 Block Size 菜单下可以为这个 RAID 0 组选择条块大小，如图 5.40 所示。至于 Block Size 参数是指整个条带的大小，还是指条带 Segment 的大小，要看厂家自己的定义。



图5.36　选择 RAID 0 模式

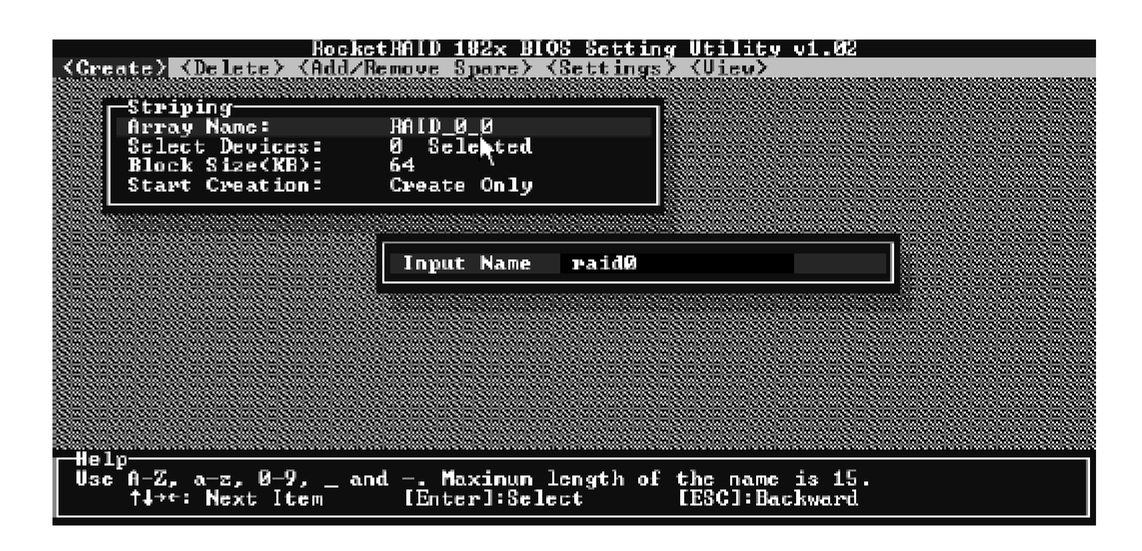


图5.37　起名“RAID 0”

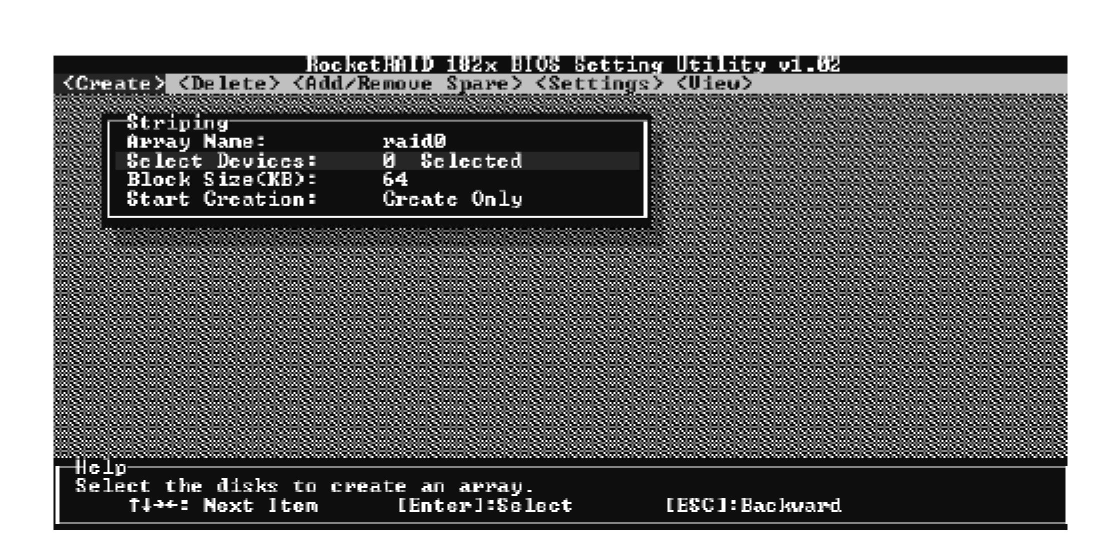


图5.38　选择磁盘(1)

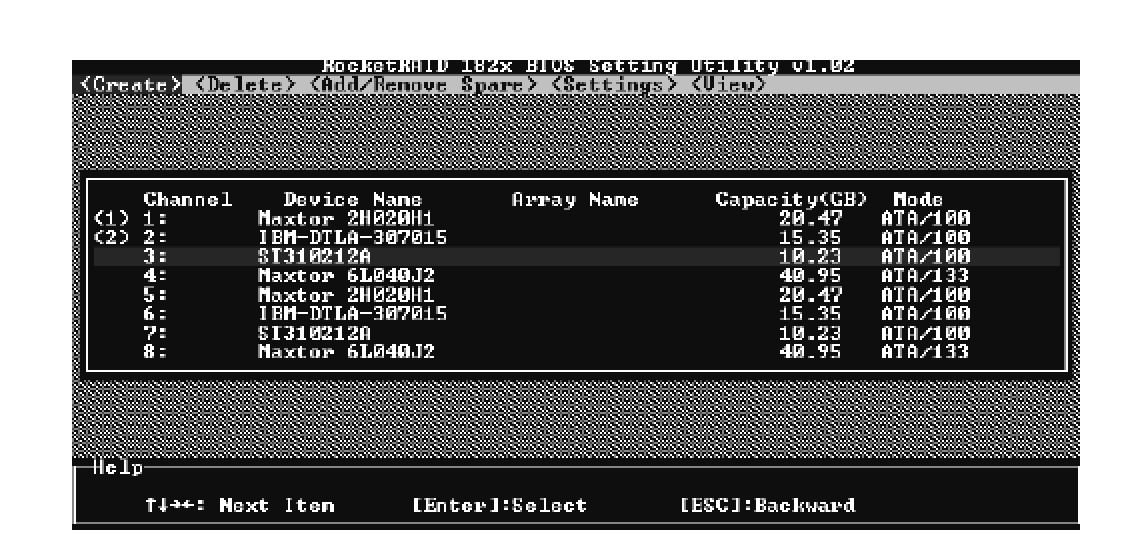


图5.39　选择磁盘(2)



图5.40　设置 Block Size

5】 选择 Start Creation，确定创建 RAID 组，如图 5.41 所示。

6】 创建完毕后，主界面中即显示出 RAID 信息，如图 5.42 所示。



图5.41　开始创建 RAID 组

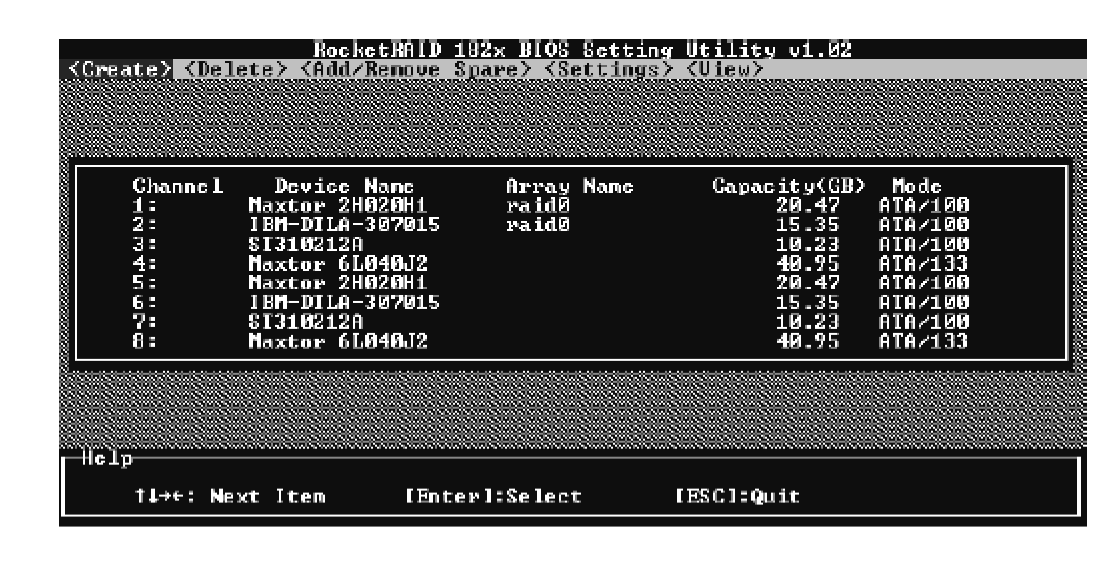


图5.42　RAID 组的信息

接下来我们继续用以上方法创建其他类型的 RAID 组。

2)　RAID 1 组的创建过程

图 5.43 所示是 RAID 1 组的创建过程，可以发现 Start Creation 中有一个 Duplication 选项，这个选项的作用是将源盘数据复制到镜像盘，而不破坏源盘数据。如果选择了 Create Only，则会破坏源盘的数据，重新创建干净的 RAID 1 组。



图5.43　创建 RAID 1 组

3)　创建一个 3 块盘组成的 RAID 5 组



在 Start Creation 菜单中有两个选项，一个为 Zero Build，另一个为 No Build，如图 5.44 所示。Zero Build 指将所有数据作废，从零开始生成数据的校验值。No Build 指不计算数据校验值，如果用户能保证 RAID 5 组中的磁盘原来是处于一致性状态的，则可以用这个选项来节约时间，否则不要选择这个选项。

如果选择 No Build 选项，则会显示警告信息，如图 5.45 所示。



图5.44 两个选项

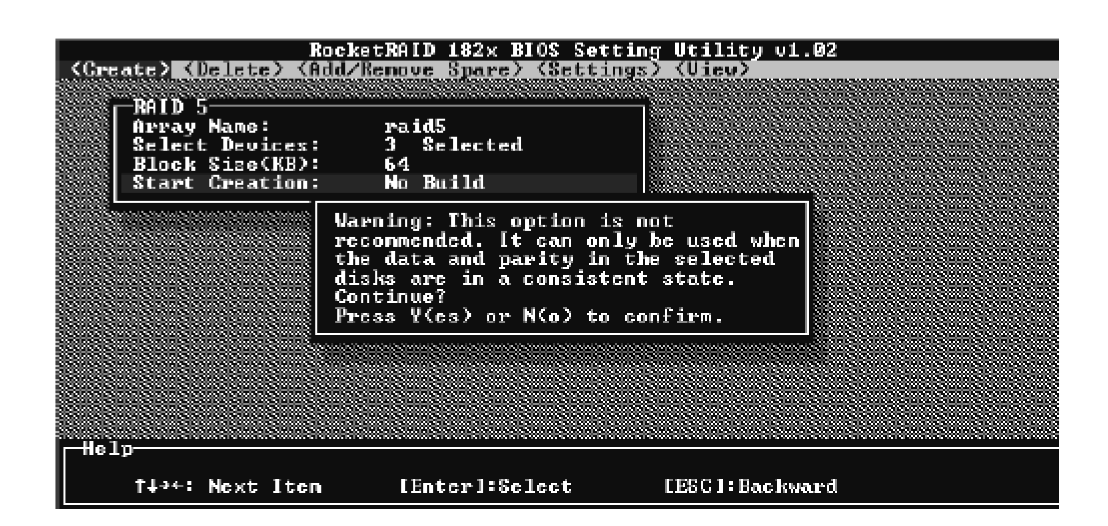


图5.45 警告信息

按 Y 键即可完成 RAID 5 组的创建。

至此，我们创建了 RAID 0、RAID 1 和 RAID 5 三个 RAID 组，如图 5.46 所示。

4) 删除 RAID 组

如果对创建的 RAID 组不满意，可以删除重建，具体操作如图 5.47 和图 5.48 所示。

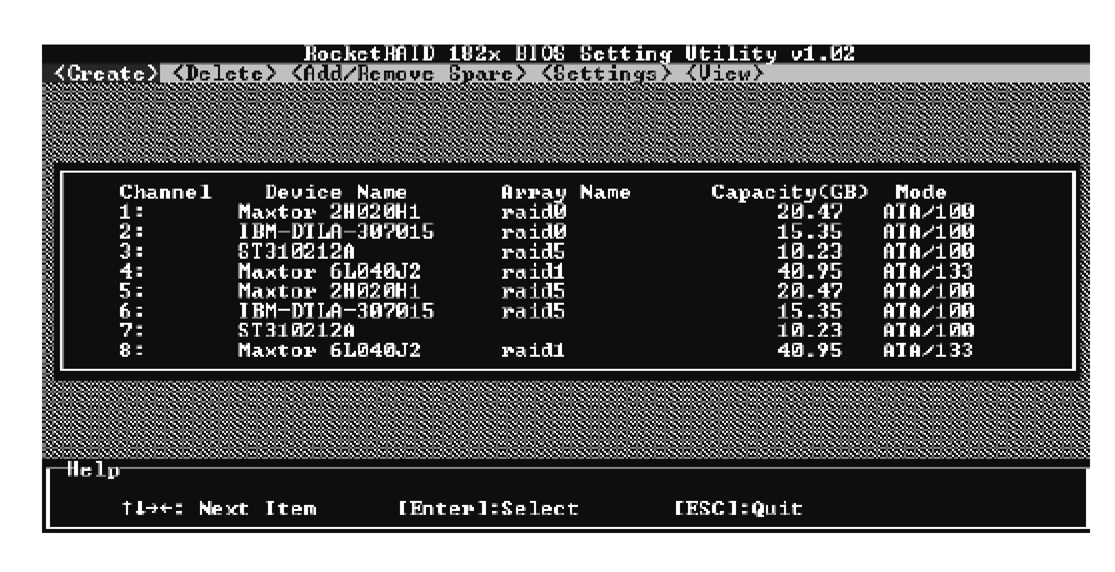


图5.46 三个 RAID 组的信息

图5.47 删除 RAID 组

5) 添加全局热备磁盘

此外，还可以添加全局热备磁盘。切换到 Add/Remove Spare 菜单，如图 5.49 所示。



图5.48 确认信息

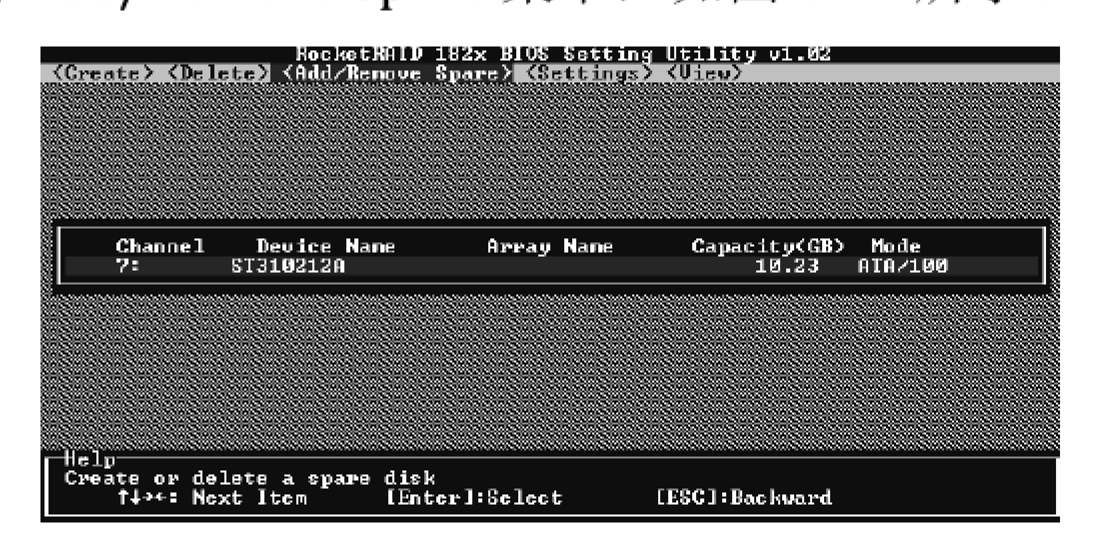


图5.49 添加全局热备盘

由于当前系统中只有一块空闲磁盘，所以我们就将这块磁盘作为全局热备磁盘，操作如图 5.50 和图 5.51 所示。如果任何 RAID 组中有磁盘损坏的话，RAID 卡将利用这块热备磁盘来顶替损坏的磁盘，将数据重新同步到这块磁盘上。

6) 设置启动标志

由于系统要从安装有操作系统的磁盘上启动，所以必须让 RAID 卡知道哪个逻辑磁盘是启动磁盘。具体设置如图 5.52、图 5.53 和图 5.54 所示。

在将 RAID 1 组形成的逻辑磁盘作为启动磁盘后，可以看见右边的“BOOT”标志。

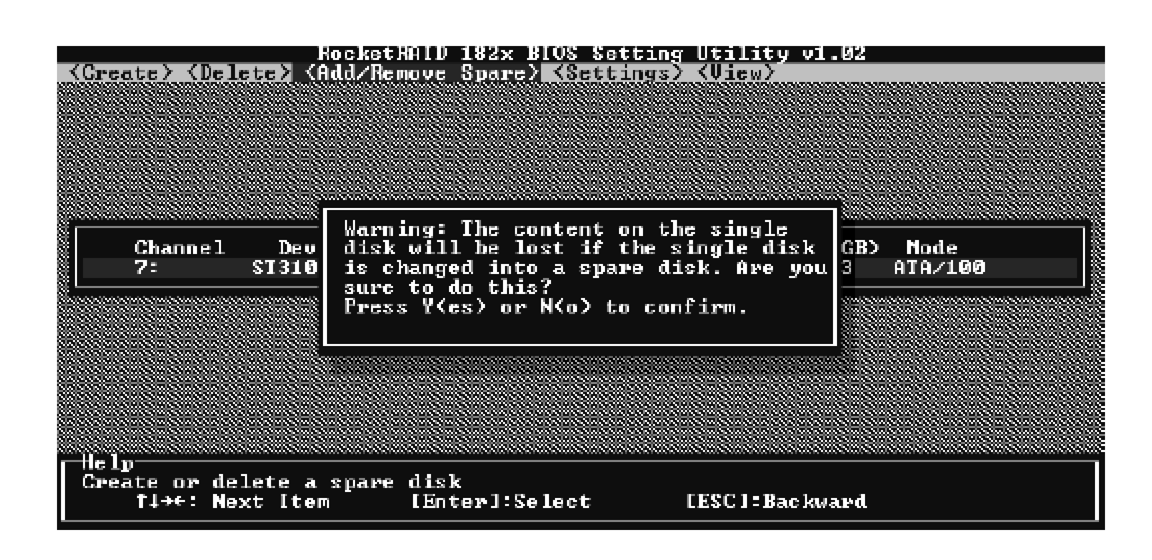


图5.50 确认信息

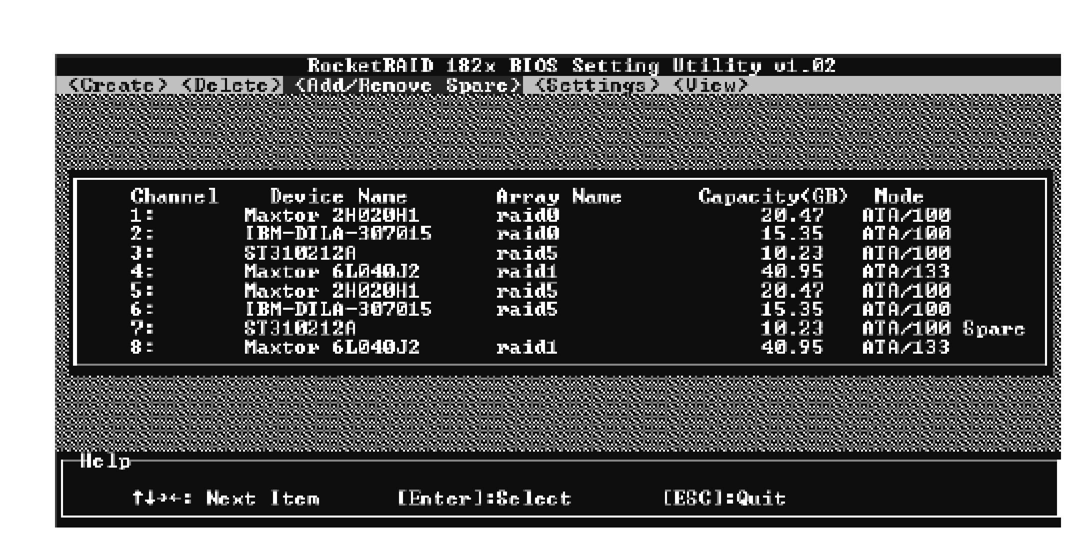


图5.51 磁盘状态



图5.52 设置启动盘(1)

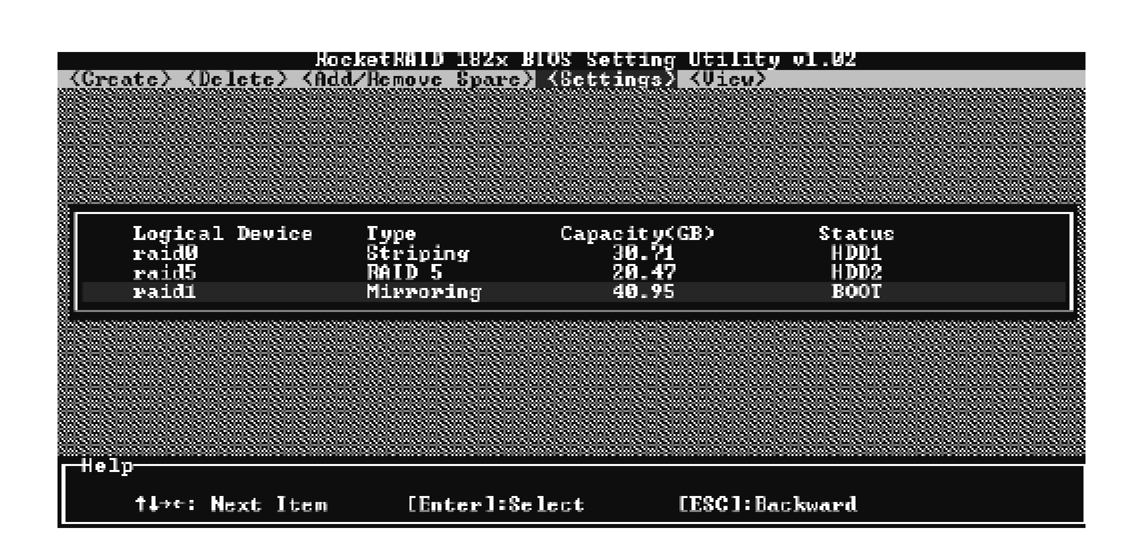


图5.53 设置启动盘(2)

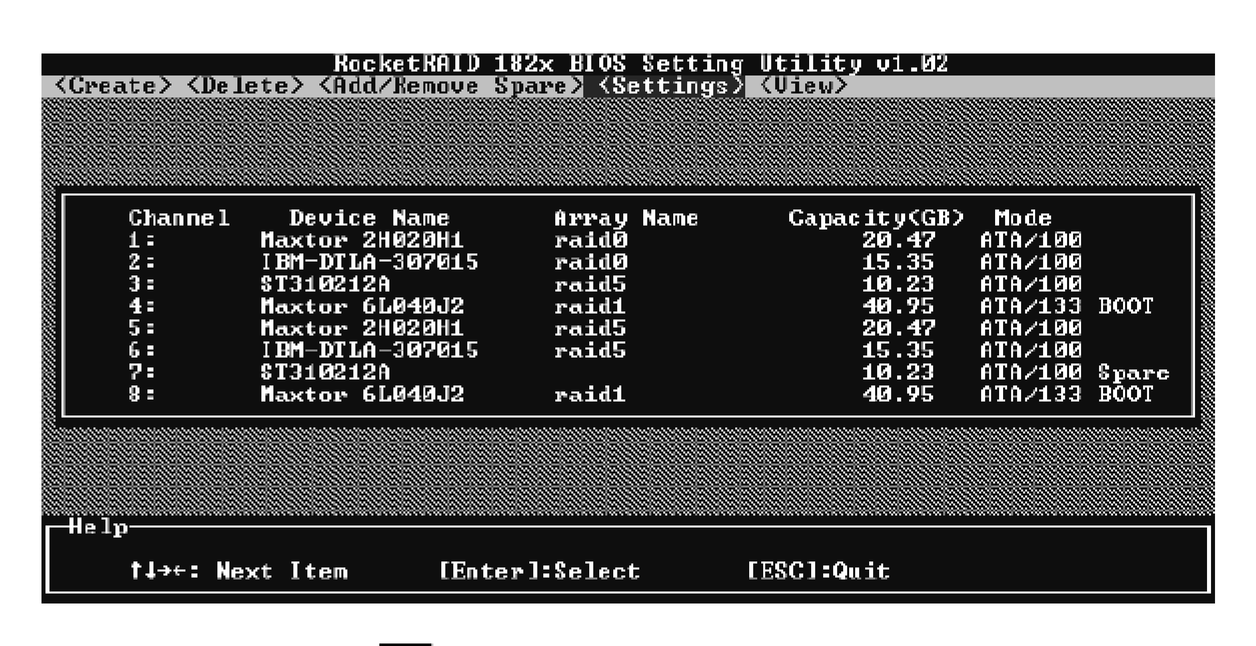


图5.54 设置启动盘(3)

7) 设置访问各个磁盘的模式参数

在 Device Mode 菜单下，可以设置访问各个磁盘的模式参数，如图 5.55 和图 5.56 所示。



图5.55 设置磁盘参数(1)

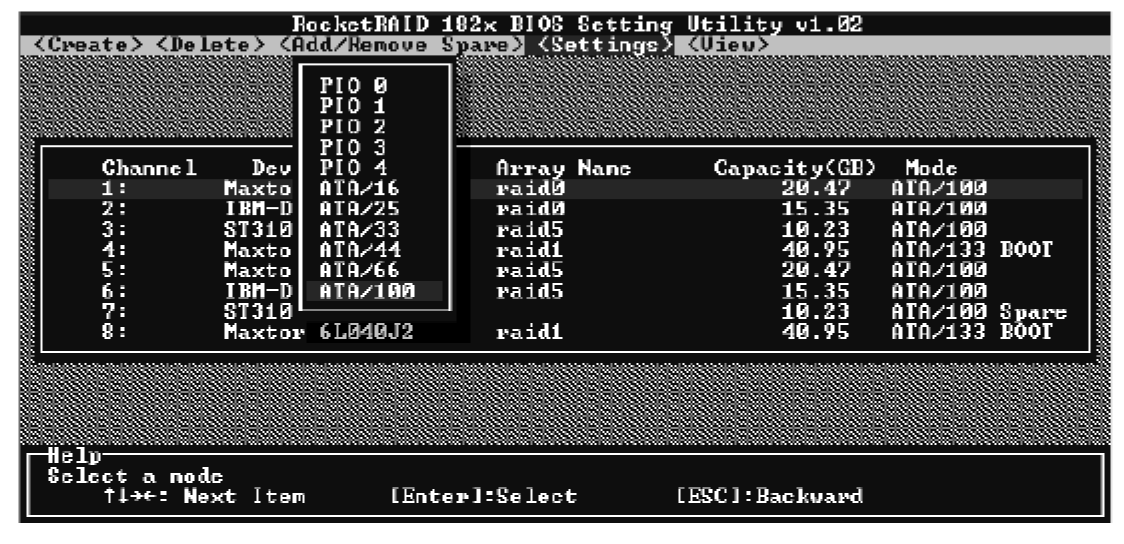


图5.56 设置磁盘参数(2)

8) 查看所有设备

在 View 菜单下，可以查看所有设备、所有 RAID 组和所有逻辑磁盘(由于这块卡不具有在 RAID 组中再次划分逻辑磁盘的功能，所以每个逻辑组只能作为一个逻辑磁盘)，如图 5.57 和图 5.58 所示。

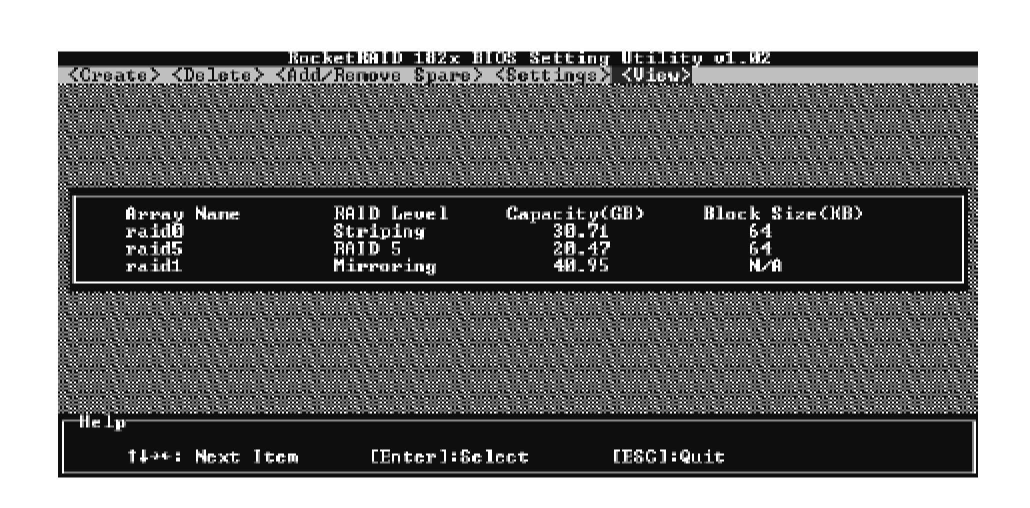


图5.57 RAID 组状态(1)



图5.58 RAID 组状态(2)

5.3 磁 盘 阵 列

RAID 卡的出现着实让存储领域变得红火起来，几乎每台服务器都标配 RAID 卡或者集成的 RAID 芯片。一直到现在，虽然磁盘阵列技术高度发展，各种盘阵产品层出不穷，但 RAID 卡依然是服务器不可缺少的一个部件。

然而，RAID 卡所能接入的通道毕竟有限，因此人们迫切希望创造一种可以接入众多磁盘、可以实现 RAID 功能并且可以作为集中存储的大规模独立设备。最终，磁盘阵列在这种需求中诞生了。

磁盘阵列的出现是存储领域的一个里程碑。关于磁盘阵列的描述，我们将在本书第 6 章中详细介绍。

5.4 实现更高级的 RAID

在 7 种 RAID 形式的基础上，还可以进行扩展，以实现更高级的 RAID。由于 RAID 0 无疑是所有 RAID 系统中最快的，所以将其他 RAID 形式与 RAID 0 杂交，将会生成更多新奇的品种。将 RAID 0 与 RAID 1 结合，生成了 RAID 10；将 RAID 3 与 RAID 0 结合，形成 RAID 30；将 RAID 5 与 RAID 0 结合，生成了 RAID 50。

5.4.1 RAID 50

图 5.59 是一个 RAID 50 的模型，RAID 30 与其类似。控制器接收到主机发来的数据之后，按照 RAID 0 的映射关系将数据分块，一部分存放于左边的 RAID 5 系统，另一部分存放在右边的 RAID 5 系统。左边的 RAID 5 系统再次按照 RAID 5 的映射关系将这一部分数据存放于 5 块磁盘中的若干块，另一边也进行相同的过程。

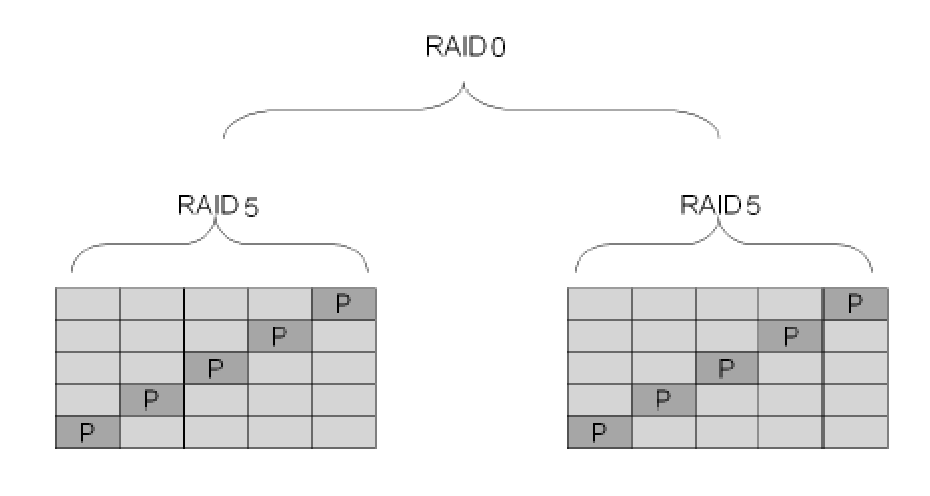


图5.59 RAID 50 模型

实际中，控制器不可能物理地进行两次运算和写 IO，这样效率很低。控制器可以将 RAID 0 和 RAID 5 的映射关系方程组合成一个函数关系方程，这样直接代入逻辑盘的 LBA，便可得出整个 RAID 50 系统中所有物理磁盘将要写入或者读取的相应 LBA 地址，然后统一向磁盘发送指令。左边的 RAID 5 系统和右边的 RAID 5 系统分别允许损坏一块磁盘而不影响数据。但是如果在任何一边的 RAID 系统同时或者先后损坏了 2 块或者更多的盘，则整个系统的数据将无法使用。

5.4.2 RAID 10 和 RAID 01

RAID 10 和 RAID 01 看起来差不多，但是本质上有一定区别。图 5.60 是一个 RAID 10 的模型。

如果某时刻，左边的 RAID 1 系统中有一块磁盘损坏，此时允许再次损坏的磁盘就剩下 2 块，也就是右边的 RAID 1 系统中还可以再损坏任意一块磁盘，而整体数据仍然是可用的。我们暂且说这个系统的冗余度变成了 2。

图 5.61 是一个 RAID 01 的模型。

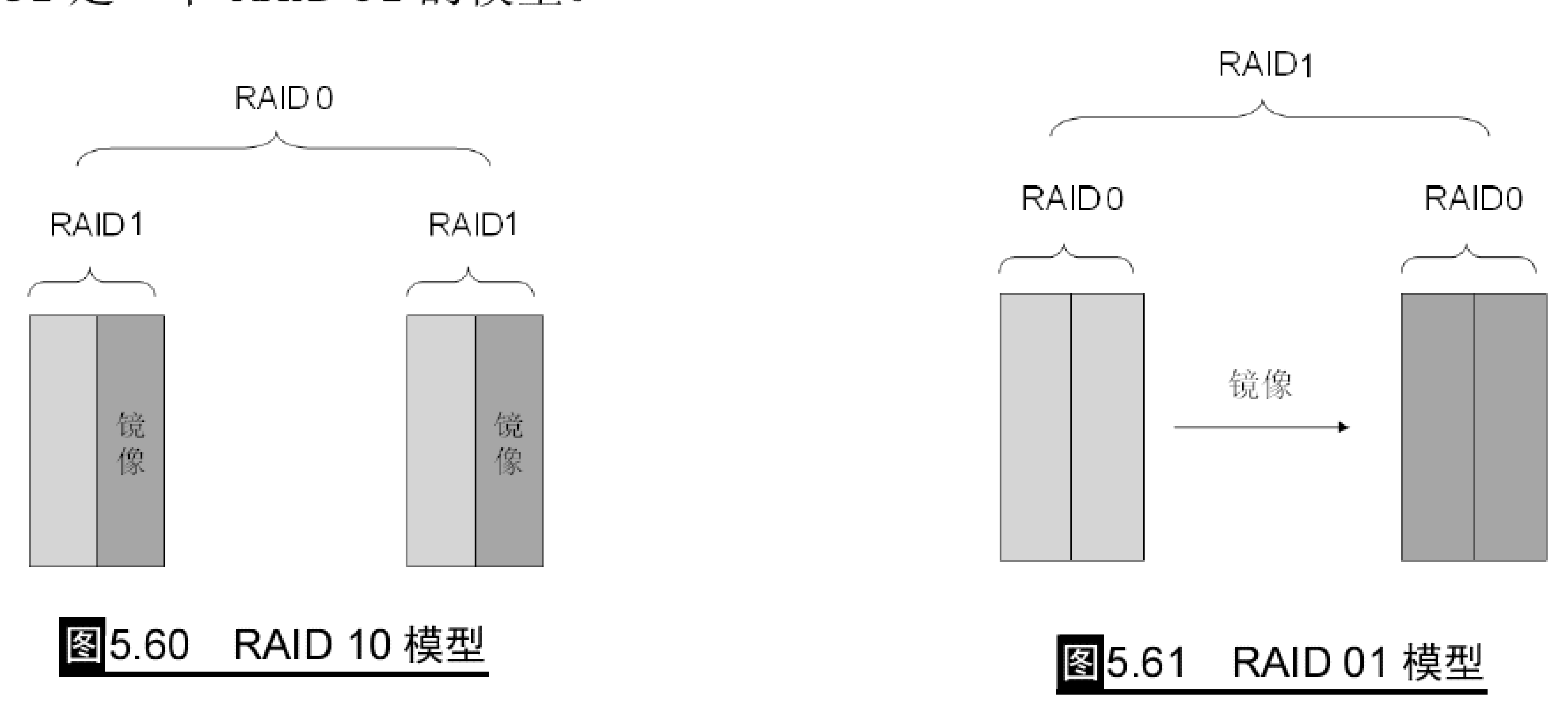


图5.60 RAID 10 模型

图5.61 RAID 01 模型

如果某时刻，左边的 RAID 0 系统中有一块磁盘损坏，此时左边的 RAID 0 系统便没有丝毫作用了。所有的 IO 均转向右边的 RAID 0 系统。而此时，仅仅允许左边剩余的那块磁盘损坏，如果右边任何一块磁盘损坏，则整体数据将不可用。所以这个系统的冗余度变成了 1，即只允许损坏特定的一块磁盘(左边 RAID 0 系统剩余的磁盘)。

综上所述，RAID 10 系统要比 RAID 01 系统冗余度高，安全性高。

5.5 虚 拟 磁 盘

话说张真人送走了七星大侠之后，面对江湖上的浮躁，有苦难言。这江湖还能出一个像七星这样的豪侠吗？难啊！七星北斗阵，多么完美的一个阵式！七星老前辈用尽毕生心血，创立了 7 种阵式，将单个磁盘组成盘阵，提高整体性能！可是很少有人能体会到这个阵式的精髓，包括创建他的七星，都不一定。张真人自从七星走后，一直处于深度悲痛之中，悔恨当初为什么没有抽时间向七星拜师学艺！如今只能守着一本老侠留下来的《七星北斗阵式》天天仔细研读，以求找到什么灵感，来继续发扬老侠的这门绝技。

……

就这样过去了 20 年。张真人已经由年轻小伙变成了稳重善思的中年人。他凭借优秀的武艺和才华，来到武当山创立了道观，并收下了 7 位徒弟，以纪念七星北斗之豪情！张真人每晚休息之前，都要对着七星北斗拜三拜。20 多年过去了，北斗的光芒依然是那么耀眼，依然看着世间纷争，昼夜交替。

这 20 年是科技飞速发展的 20 年。铁匠们的技艺提高很快，新技术不断被创造出来。大容量、高速度的磁盘在地摊卖 10 文钱一斤。

某天张老道下山溜达，发现地摊上的磁盘品质还不错，比 20 年前的货强太多了，顺手就买了 50 斤回去。点了点，足足 50 块。他让他的 7 位徒弟，分别按照七星阵摆上各种阵形，来捣鼓这 50 块硬盘，7 位徒弟早就对七星阵烂熟于心，把这 50 块磁盘捣鼓得非常顺。张老道频频点头，心里想着："嗯，应了那句话啊。长江后浪推前浪，一代新人换旧人！"摆弄了一阵之后，徒弟们都累了。这次格外地累，不禁都坐在地上休息。老道把眼一瞪，"嗯忒！！！！年轻人，不好好练功，不准偷懒！"徒弟们上前道："师父，不是我们偷懒，这次您买的磁盘和以前的不一样。我们在出招的时候，就是在"化龙"这一招的时候特别吃力。这条龙太大，不好操控。"老道一看，果然，这 50 块磁盘每块足有 1TB 大，50 块就是 50TB。"霍霍，20 年前一块磁盘最多也就是 50MB，没想到啊！"

这天晚上，老道用完粗茶淡饭之后，遥望北斗，心想七星老侠在天上不知道看见此情此景，会给我什么启示呢？20 年前，用此阵式生成的虚拟磁盘，大小也不过几 G，而如今已经达到了 T 级别，也难怪我那些徒儿们会吃不消。怎么办呢？需要把这以 T 论的虚拟磁盘再次划分开来，划分成多条"小龙"，这样就可以灵活操控了。而且针对目前的磁盘超大的容量，完全可以在一个阵中同时应用多种阵式。比如让我 7 位徒弟，其中 3 人摆出 RAID 0 阵式，另外 4 人同时摆出 RAID 5 阵式，共同出招。每个阵式生成的虚拟"龙盘"，把它划分成众多小的"龙盘"，这样对外不但我们的威力没有减少，而且可以灵活运用，让敌人不知道我们到底有几个人。

张老道决定将大龙盘划分成小龙盘，这事十分好办，只需体现在"心中"就可以了。只要你心中有数，那些物理磁盘的哪部分区域属于哪个小龙盘，就完全可以对外通告了。老道称这种技术为逻辑盘技术。

5.5.1 RAID 组的再划分

实际中，比如用 5 块 100GB 的磁盘做了一个 RAID 5，那么实际数据空间可以到 400GB，剩余 100GB 空间是校验空间。如果将这 400GB 虚拟成一块盘，不够灵活。且如果 OS 不需要这么大的磁盘，就没法办了。所以要再次划分这 400GB 的空间，比如划分成 4 块 100GB 的逻辑磁盘。而这逻辑盘虽然也是 100GB，但是并不同于物理盘，向逻辑盘写一个数据会被 RAID 计算，而有可能写向多块物理盘，这样就提升了性能，同时也得到了保护。纵使 RAID 组中坏掉一块盘，操作系统也不会感知到，它看到的仍然是 100GB 的磁盘。

5.5.2 同一通道存在多种类型的 RAID 组

不仅如此，老道还想到了在一个阵式中同时使用多种阵法的方式。

实际中，假设总线上连接有 8 块 100GB 的磁盘，我们可以利用其中的 5 块磁盘来做一个 RAID 5，而后再利用剩余的 3 块磁盘来做一个 RAID 0，这样，RAID 5 的可用数据空间为 400GB，校验空间为 100GB，RAID 0 的可用数据空间为 300GB。而后，RAID 5 和 RAID 0 各自的可用空间，又可以根据上层 OS 的需求，再次划分为更小的逻辑磁盘。这样就将七星北斗阵灵活地运用了起来，经过实践的检验，这种应用方法得到了巨大的推广和成功。

张老道给划分逻辑盘的方法取名为巧化神龙，将同一个阵中同时使用多种阵式的方法叫做神龙七变。

5.5.3 操作系统如何看待逻辑磁盘

目前各种 RAID 卡都可以划分逻辑盘，逻辑盘大小任意设置。每个逻辑盘对于 OS 来说都认成一块单独的物理磁盘。这里不要和分区搞混，分区是 OS 在一块物理磁盘上做的再次划分。而 RAID 卡提供给 OS 的，任何时候，都是一块或者几块逻辑盘，也就是 OS 认成的物理磁盘。而 OS 在这个磁盘上，还可以进行分区、格式化等操作。

5.5.4 RAID 控制器如何管理逻辑磁盘

我们来看一下 RAID 卡对逻辑磁盘进行再次划分的具体细节。既然要划分，就要心中有数，比如某块磁盘的某个区域，划分给哪个逻辑盘用，对应逻辑盘的 LBA 地址是多少，这块磁盘的 RAID 类型是什么等。而这些东西不像 RAID 映射那样根据几个简单的参数就能确定，而且对应关系是可以随时变化的，比如扩大和缩小、移动等。所以有必要在每块磁盘上保留一个区域，专门记录这种逻辑盘划分信息，RAID 类型以及组内的其他磁盘信息等，这些信息统称为 RAID 信息。不同厂家、不同品牌的产品实现起来不一样，SNIA 委员会为了统一 RAID 信息的格式，专门定义了一种叫做 DDF 的标准，如图 5.62 所示。

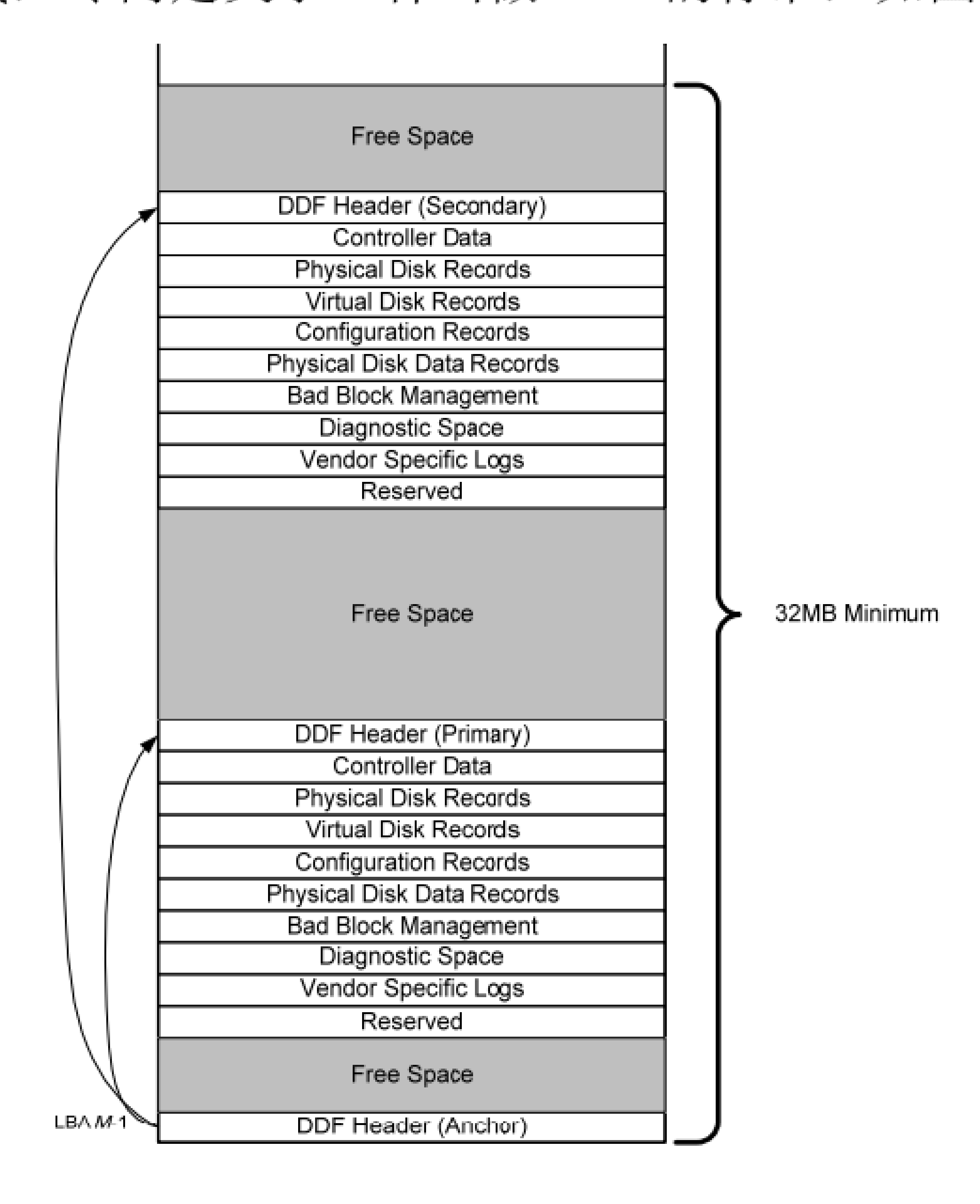


图5.62 DDF 布局图

图 5.63 所示的是微软和 Veritas 公司合作开发的软 RAID 在磁盘最末 1MB 空间创建的数据结构。有了这个记录，RAID 模块只要读取同一个 RAID 子系统中每块盘上的这个记录，就能够了解 RAID 信息。即使将这些磁盘打乱顺序，或者拿到其他支持这个标准的控制器上，也照样能够认到所划分好的逻辑盘等所有需要的信息。

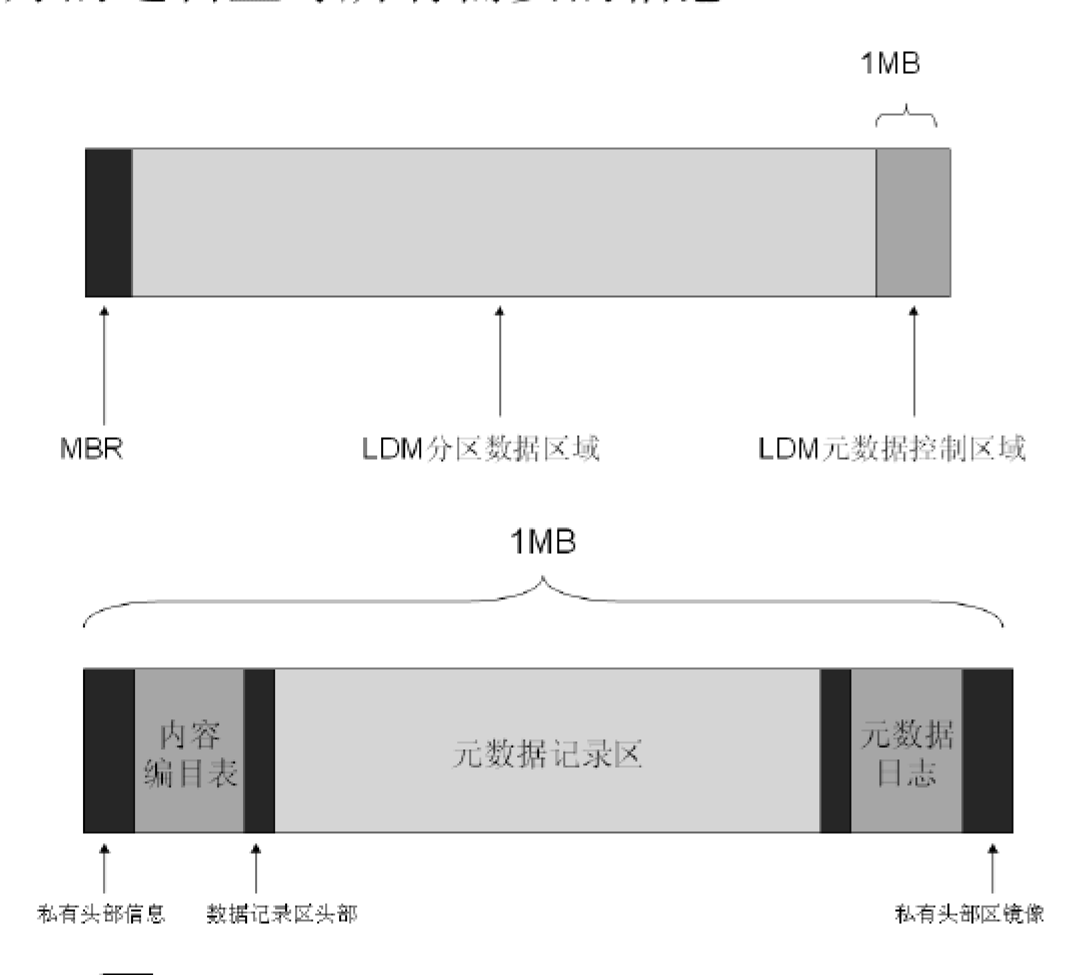


图5.63 Windows 系统中的动态磁盘信息

RAID 卡可以针对总线上的某几块磁盘做一种 RAID 类型，然后针对另外的几块磁盘做另一种 RAID 类型。一种 RAID 类型中包含的磁盘共同组成一个 RAID Group，简称 RG。逻辑盘就是从这个 RG 中划分出来的，原则上逻辑盘不能跨 RG 来划分，就是说不能让一个逻辑盘的一部分处于一个 RG，另一部分处于另一个 RG。因为 RG 的 RAID 类型不一样，其性能也就不一样，如果同一块逻辑盘中出现两种性能，对上层应用来说不是件好事，比如速度可能会忽快忽慢等。

张真人推出了这两门绝技之后，在江湖上引起了轩然大波。大家争相修炼，并取得了良好的效果。一时间，江湖上几乎人人都练了张真人这两种功夫。而且各大门派已经将七星北斗阵以及张真人的功夫作为各派弟子必须掌握的基本功。

近水楼台先得月。武当七子当然已经把功夫练到了炉火纯青的地步。老道非常欣慰。他相信七星侠在天之灵倘若看到了这阵式被拓展，一定也会感到欣慰的。

5.6 卷管理层

老道创立这两种功夫的兴奋，很快就被一个不大不小的问题给吹得烟消云散。这个问题就是一旦逻辑盘划分好之后就无法改变，要改变也行，上面的数据就得全部抹掉，这是让人无法容忍的。比如已经做好了一个 100GB 的逻辑盘，但是用了 2 年以后，发现数据越来越多，已经盛不下了。但又不能放到别的磁盘，因为受上层文件系统的限制，一个文件不可能跨越多个分区来存放，更别提跨越多个磁盘了。如果有一个文件已经超过了 100GB，那么谁也无力回天，只能重新划分逻辑盘。数据怎么办？这问题遇不到则已，遇到了就是死路一条。江湖上已经有不少生意人因为这个问题而倾家荡产，他们无奈之余，准备联合起来到武当恳求张老道想一个办法，以克服这个难关，好让他们东山再起。张老道对他们

的遭遇深感同情，同时也责怪自己当初疏忽了这个问题。于是他当众许下承诺，3 个月之后，来武当取解决办法。

5.6.1 有了逻辑盘就万事大吉

1. 踏破铁鞋无觅处——寻找更加灵活的磁盘卷管理方式

其实张真人许下 3 个月的时间，他自己也是豪无把握。但是为了平息众怒，也只能冒险赌一次了！送走众人之后，张老道就开始天天思考解决这个问题的办法。他想，到底怎么样才能让使用者运用自如呢？如果一开始就给他划分一个 100GB 的逻辑盘，如果数据盛不下了，此时把其他磁盘上未使用的空间挪一部分到这个逻辑盘，岂不是就可以了么？

可以是可以，但从 RAID 卡设置里增加或减少逻辑盘容量很费功夫。在 RAID 卡里增加这种代码，修炼成本很高，而且即使实现了，主机也不能立即感应到容量变化。即使感应到了，也不能立即变更。对于 Windows 系统来说，必须将其创建为新的分区。想要合并到现有分区，必须用第三方分区表调整工具在不启动操作系统的情况下来修改分区表才行。再者，其上的文件系统不一定会跟着扩大，NTFS 这种文件系统不能动态张缩，也必须在不启动操作系统的条件下用第三方工具调整。这方法对一些要求不间断服务的应用服务器并不适用。

老道想到这里，觉得至少是已经找到了一种解决办法，虽然不是很方便，需要重启主机，之后再在 RAID 卡中更改配置。更改完毕后，可能还要重启一次，然后进入系统，系统才能认出新容量的磁盘。而 OS 就算正确认出了新增的磁盘容量，由于分区表没有改变，新增容量不属于任何一个分区，还是不能被使用，所以还需要手动修改分区表。太复杂、太麻烦了，能否找一种方便快捷的方法呢？

2. 得来全不费工夫——来源于现实的刺激

话说冬至这天，天上飘着雪花。武当是张灯结彩，喜气洋洋！这天是张真人的 70 大寿！江湖各大门派及各路英雄纷纷前来拜寿。武当上下忙得是不亦乐乎！就说包饺子吧，一会儿面不够了去和面，一会儿水不够了去挑水。张真人是往来作揖，笑迎来宾。厨房则加紧和面，由于厨房则空间太小，所以和好的面被运往各个分理点处，那里有小道负责擀皮包饺子。张真人看着眼前这小老道跑来跑去的多少回了，就纳闷了，所以跟着去看看怎么回事。一看才知道，弄了半天是往各处运面团呢！觉得挺好笑的，也没当回事。等大家都差不多到齐了，共同给老道祝了寿，然后就上饺子了。张老道看着碗里一个个的饺子，再想想刚才那面团的事，心里突然一动！于是当众宣布，一个月前自己承诺的约定过不了几天就会实现了！众豪杰是一片掌声！



张老道到底想到了什么呢？原来，他想起了小道包饺子和面的情形。厨房和了一大团面，下面随用随取。不够了，割一块揉进去就行了，或者掰下一块来放着下次用。这不正解决了一个月前大家所头疼的问题么？RAID 控制器给和好了几团面(逻辑盘)，放那由自己看着用，哪不够了就掰块补上。必须实现这样一种像掰面团一样灵活的管理层，才能最终解决使用中出现的问题。是啊，说得简单，可是具体要做却不是那么容易的。

当天晚上，老道睡觉的时候就一个劲地想，在 RAID 控制器上掰面，以前也分析过了，不合适，那么在哪里掰呢？RAID 控制器给你和了几斤面，你就得收着，不要也不行。但是面收着了，你可以自己掰呀，是啊，自己掰。那么就是说，RAID 控制器提交给 OS 的逻辑磁盘。应该可以掰开，或者揉搓到一块儿去，可以想怎么揉搓就怎么揉搓。这功能如果能通过在操作系统上运行一层软件来实现的话，不但灵活，而且管理方便！想到这，老道心里有了底。

第二天，老道就让徒弟们按照他写的口诀来实现他这个想法，大获成功！RAID 控制器是硬件底层实现 RAID，实现逻辑盘，所以操作起来不灵活。如果在 OS 层再把 RAID 控制器提交上来的逻辑盘(OS 会认成不折不扣的物理磁盘)加以组织、再分配，就会非常灵活。因为 OS 层上运行的都是软件，完全靠 CPU 来执行，而不用考虑太多的细节。张老道立即将这种新的掌法公布天下，称作神仙驾龙！

5.6.2 卷管理层

实际中，有很多基于这种思想的产品，这些产品都有一个通用的名称，叫做卷管理器(Volume Manager，VM)。比如微软在 Win2000 中引入的动态磁盘，就是和 Veritas 公司合作开发的一种 VM，称为 LDM(逻辑磁盘管理)。Veritas 自己的产品 Veritas Volume Manager(VxVM)和广泛用于 Linux、AIX、HPUX 系统的 LVM(Logical Volume Manager)，以及用于 Sun Solaris 系统的 Disk Suite，都是基于这种在 OS 层面，将 OS 识别到的物理磁盘(可以是真正的物理磁盘，也可以是经过 RAID 卡虚拟化的逻辑磁盘)进行组合，并再分配的软件。它们的实现方法大同小异，只不过细节方面有些差异罢了。

这里需要重点讲一下 LVM，因为它的应用非常普遍。LVM 开始是在 Linux 系统中的一种实现，后来被广泛应用到了 AIX 和 HPUX 等系统上。

- PV：LVM 将操作系统识别到的物理磁盘(或者 RAID 控制器提交的逻辑磁盘)改了个叫法，叫做 Physical Volume，物理卷(一块面团)。
- VG：多个 PV 可以被逻辑地放到一个 VG 中，也就是 Volume Group 卷组。VG 是一个虚拟的大存储空间，逻辑上是连续，尽管它可以由多块 PV 组成，但是 VG 会将所有的 PV 首尾相连，组成一个逻辑上连续编址的大存储池，这就是 VG。
- PP：也就是 Physical Partition(物理区块)。它是在逻辑上再将一个 VG 分割成连续的小块(把一大盆面掰成大小相等的无数块小面团块)。注意，是逻辑上的分割，而不是物理上的分割，也就是说 LVM 会记录 PP 的大小(由几个扇区组成)和 PP 序号的偏移。这样就相当于在 VG 这个大池中顺序切割，如果设定一个 PP 大小为 4MB，那么这个 PP 就会包含 8192 个实际物理磁盘上的扇区。如果 PV 是实际的一块物理磁盘，那么这些扇区就是连续的。如果 PV 本身是已经经过 RAID 控制器虚拟化而成的一个 LUN，那么这些扇区很有可能位于若干条带中，也就是说这 8192 个扇区物理上不一定连续。
- LP：PP 可以再次组成 LP，即 Logical Partition(逻辑区块)。逻辑区块是比较难理解的一个东西，一个 LP 可以对应一个 PP，也可以对应多个 PP。前者对应前后没什么区别。后者又分两种情况，一种为多个 PP 组成一个大 LP，像 RAID 0 一样；另一种是一个 LP 对应几份 PP，这几份 PP 每一份内容都一样，类似于 RAID 1，

多个 PP 内容互为镜像，然后用一个 LP 来代表它们，往这个 LP 写数据，也就同时写入了这个 LP 对应的几份 PP 中。

- LV：若干 LP 再经过连续组合组成 LV(Logical Volunme，逻辑卷)，也就是 LVM 所提供的最终可用来存储数据的单位。生成的逻辑卷，在主机看来还是和普通磁盘一样，可以对其进行分区、格式化等。



有人问了，一堆面团揉来揉去，最终又变成一堆面团了，你这是揉面还是做存储呢？

确实，面团最终还是面团。但是此面团非彼面团。最终形成的这个 LV，它的大小可以随时变更，也不用重启 OS，你想给扩多大就扩多大，前提是面盆里面还有被掰开备用的 PP。而且，只要盆里面有 PP，你就可以再创建一个 LV，也就是再和一团面，LV 数量足够用的。如果不增加卷管理这个功能，那么 RAID 卡提交上来多少磁盘，容量多大就是多大，不能在 OS 层想改就改、为所欲为。而卷管理就提供了这个为所欲为的机会，让你随便和面团。

LVM 看起来很复杂，其实操作起来很简单。创建 PV，将 PV 加入 VG，在 VG 中再创建 LV，然后格式化这个 LV，就可以当成一块普通硬盘使用了。容量不够了，还可以随便扩展，岂不快哉？LVM 一个最大的好处就是生成的 LV 可以跨越 RAID 卡提交给 OS 的物理磁盘(逻辑盘)。这是理所当然的，因为 LVM 将所有物理盘都搅和到一个大面盆中了，当然就可以跨越物理盘了。

5.6.3 Linux 下配置 LVM 实例

下面我们以 RedHat Enterprise Linux Server 4 Update 5 操作系统为例，给大家示例一下 LVM 的配置过程。

1】 在操作系统安装过程中，选择手动配置磁盘管理，如图 5.64 所示。

2】 可以看到，这台机器共有 8 块物理磁盘，每块的容量为 1GB，如图 5.65 所示。

3】 首先，需要定义一个/boot 分区，这个分区是用来启动基本操作系统内核的，所以这块空间不能参与 LVM。我们选择从第一块硬盘(sda)划分出 20 个磁道的空间用来作为/boot 分区。这块空间也就成了 sda1 设备，如图 5.66 所示。

4】 接下来，对于 sda2、sdb、sdc、sdd、sde、sdf、sdg、sdh 所有这些剩余的磁盘或者分区，就可以将它们配置成 LVM 的 PV(物理卷)。选中每个磁盘或者分区，单击 Edit 按钮。在 File System Type 下拉列表框中，选择 physical volume(LVM)选项，表示将这个硬盘或者分区配置成 LVM 的 PV。PV 可以任意设定大小，只要编辑 End Cylinder 文本框中的值即可。剩余空间可以继续作为 PV 再次分配。对每个磁盘都进行上述操作，如图 5.67 所示。

5】 操作完成后，信息栏中显示所有磁盘和 sda2 分区都已被配置成为 PV，如图 5.68 所示。

6】 单击 LVM 按钮会出现如图 5.69 所示的对话框。这一步就是创建 VG(Volume Group，卷组)的过程。可以将 PV 进行任意组合，组合后的 PV 就形成了 VG。

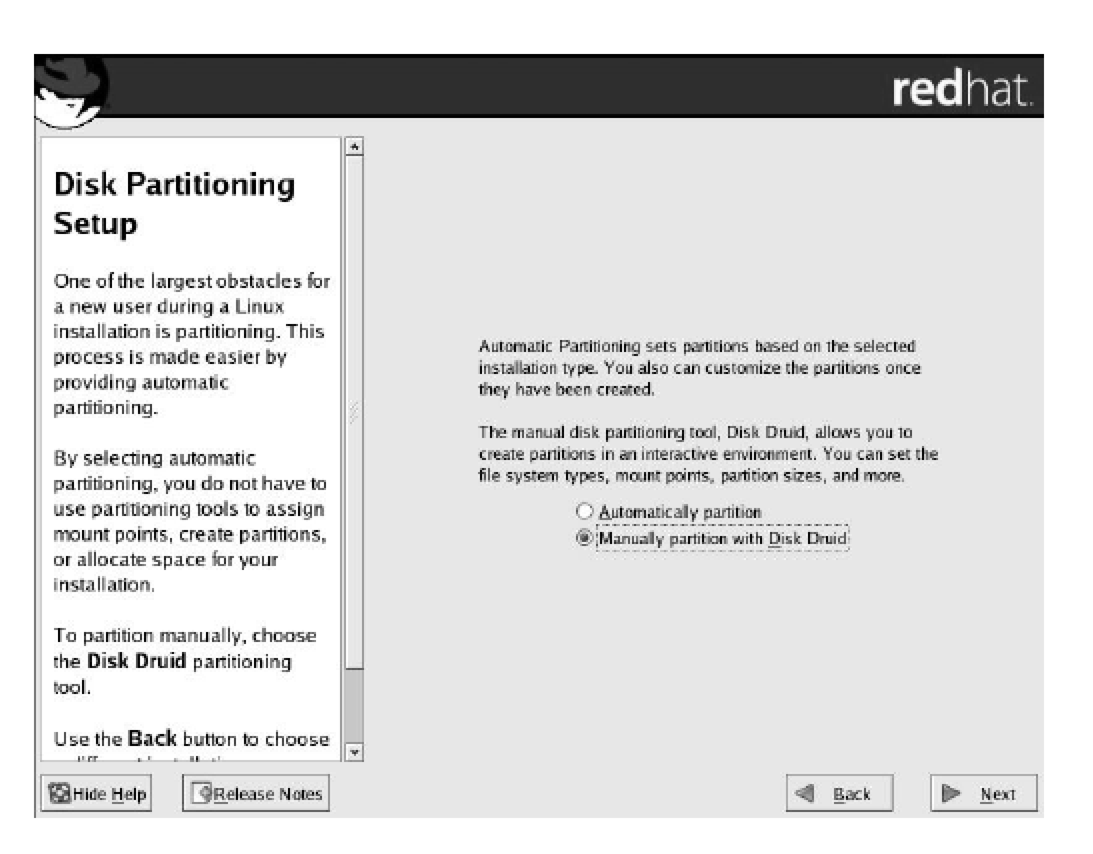


图5.64　选择手动管理

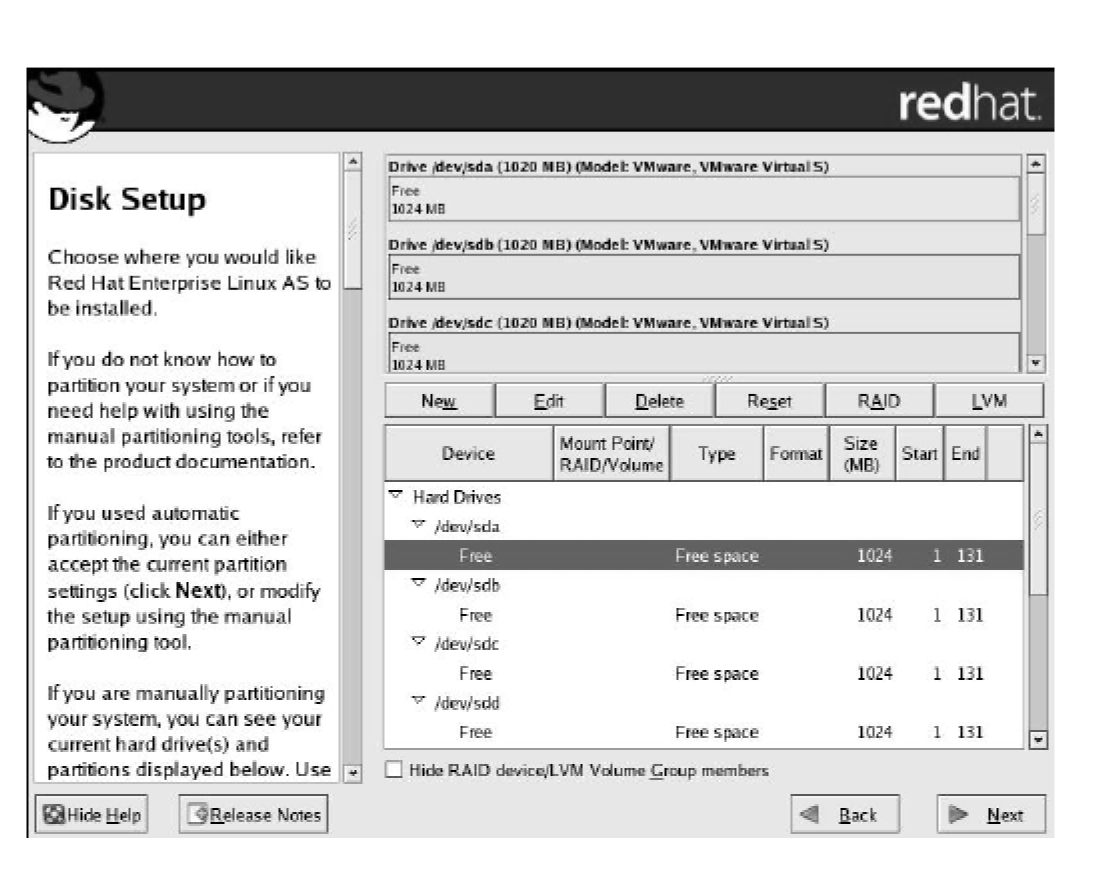


图5.65　磁盘列表

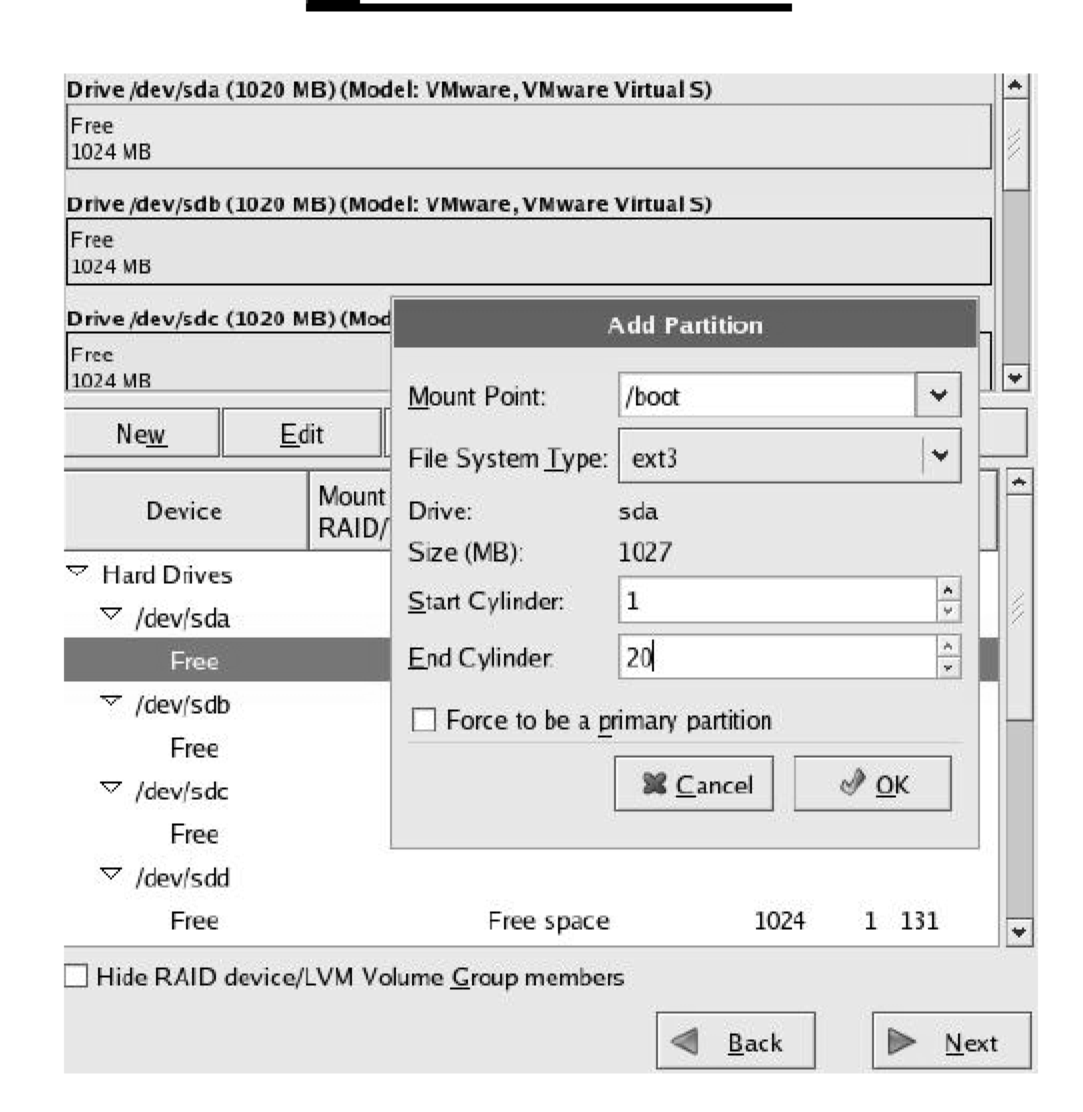


图5.66　创建/boot 分区

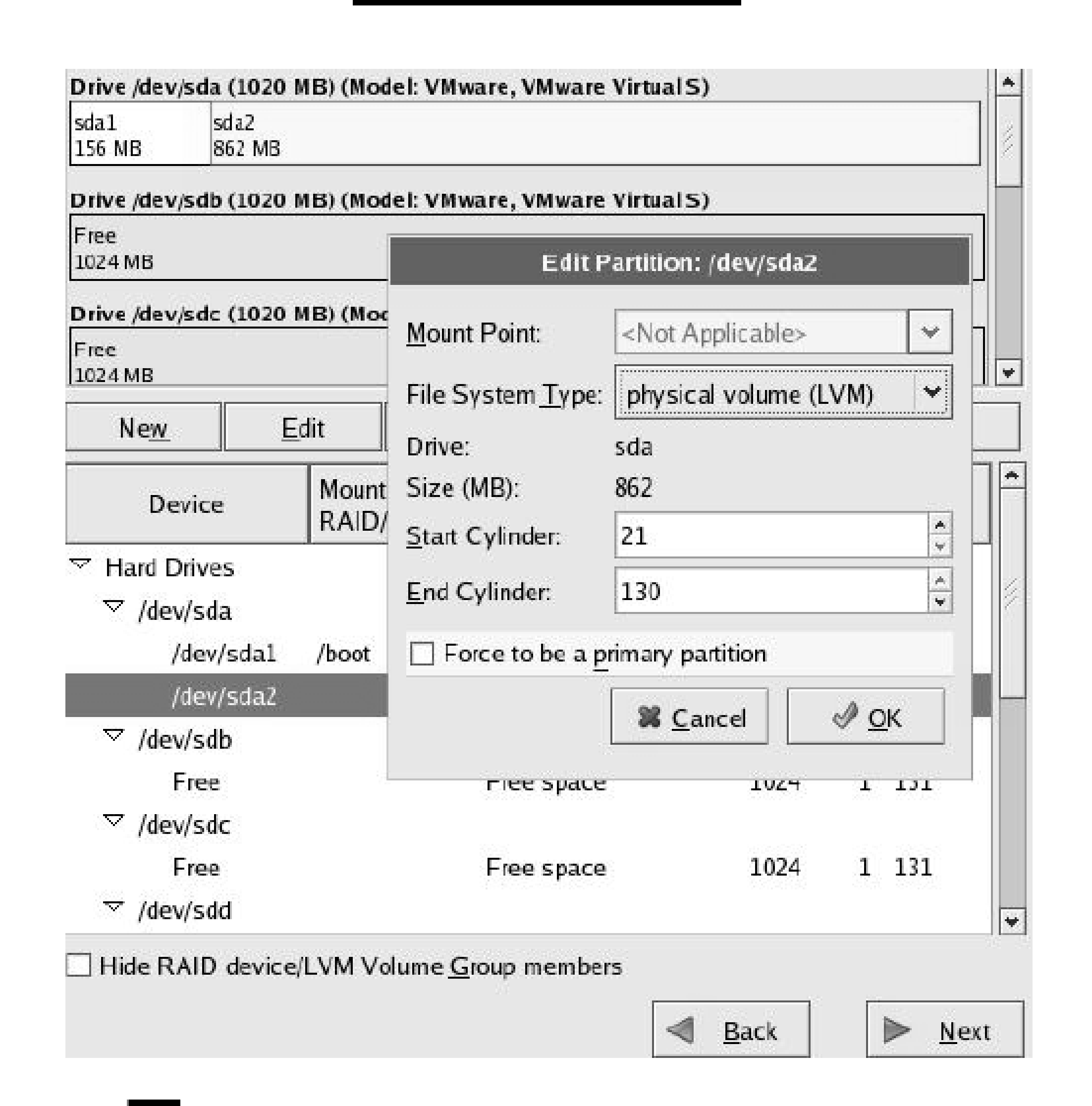
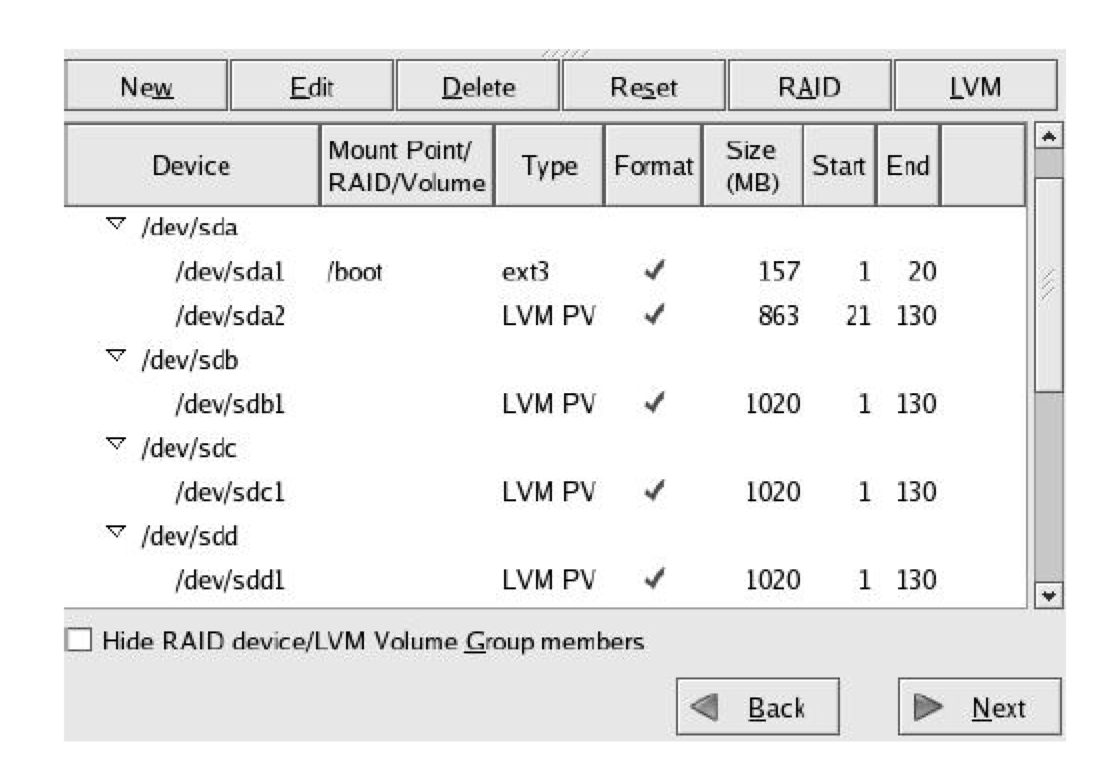


图5.67　设置磁盘类型为 LVM 管理状态



Device	Mount Point/ RAID/Volume	Type	Format	Size (MB)	Start	End
/dev/sda						
/dev/sda1	/boot	ext3	✓	157	1	20
/dev/sda2		LVM PV	✓	863	21	130
/dev/sdb						
/dev/sdb1		LVM PV	✓	1020	1	130
/dev/sdc						
/dev/sdc1		LVM PV	✓	1020	1	130
/dev/sdd						
/dev/sdd1		LVM PV	✓	1020	1	130

图5.68　磁盘状态

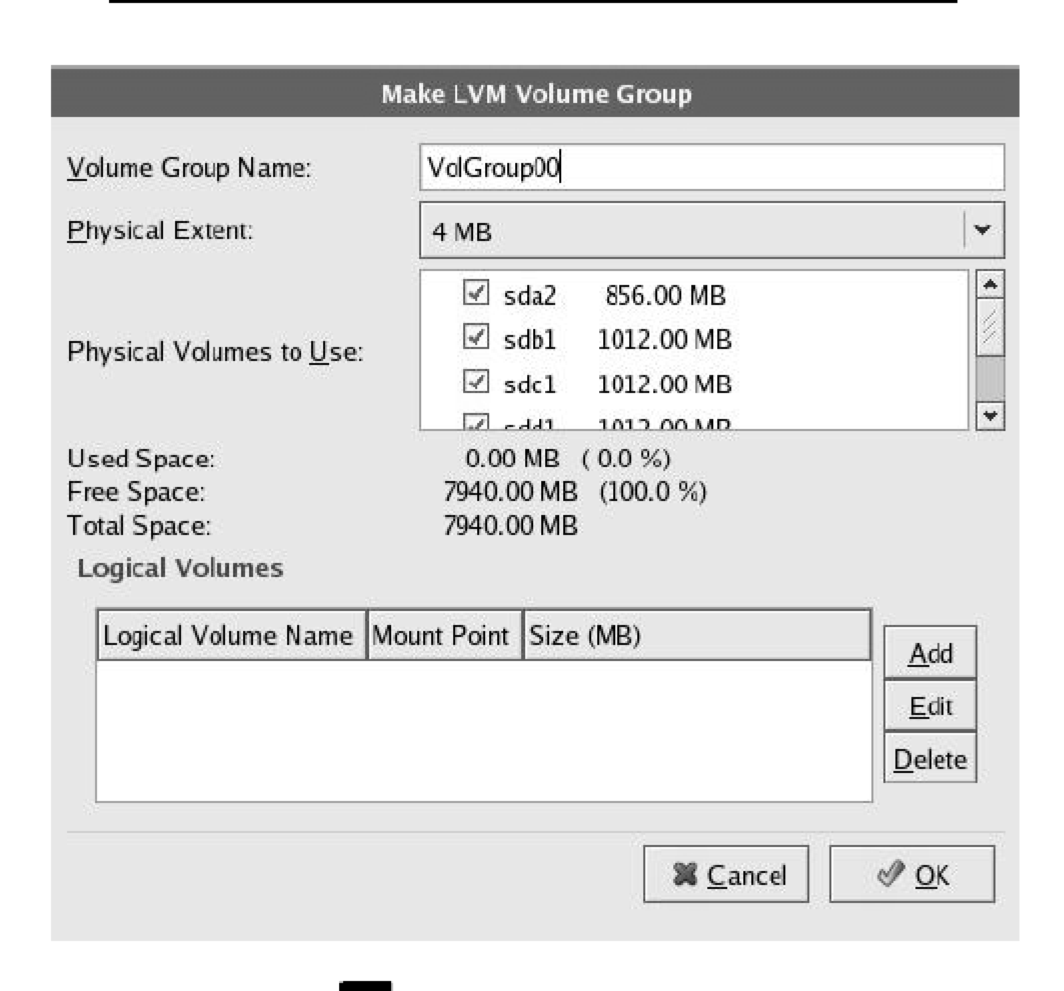


图5.69　创建卷组

7】 这里我们做一个名为“VolGroup00”的卷组，其包含 sda2 和 sdb1 两个 PV。在 Physical Extent 下拉列表框中，可以选择这个卷组对应的磁盘空间的最小分配单位(在 AIX 的 LVM 中，这个最小单位成为 Physical Partition，即 PP)。然后单击下方的 Add 按钮，从这个大的卷组空间中再次划分逻辑卷，即 LV。下面创建一个大小为 1000MB 的逻辑卷 LogVol00，并且用 ext3 文件系统将这个卷格式化，并挂载到/home 目录下，如图 5.70 所示。

8】 然后将 VolGroup00 卷组中剩余的空间，全部分配给一个新的 LV，即 LogVol01，用 ext3 文件系统格式化，并挂载到/tmp 目录下，如图 5.71 所示。

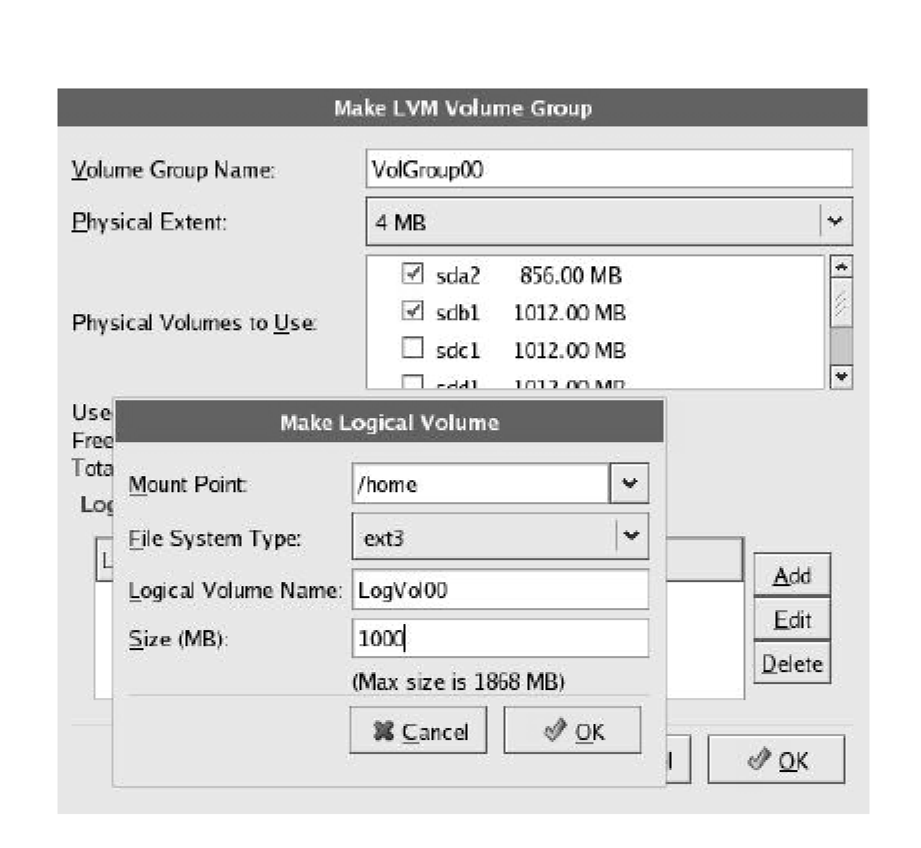


图5.70 创建 LV 并挂载

图5.71 创建 LV 并挂载

9】 将剩余的 sdc1、sdd1、sde1、sdf1、sdg1、sdh1 这几块 PV 全部分配给一个新的卷组 VolGroup01，并且在卷组中创建一个逻辑卷 LogVol00，大小为整个卷组的大小，用 ext3 文件系统格式化，并挂载到/目录下，如图 5.72 所示。

10】 配置完成后的状态如图 5.73 所示。

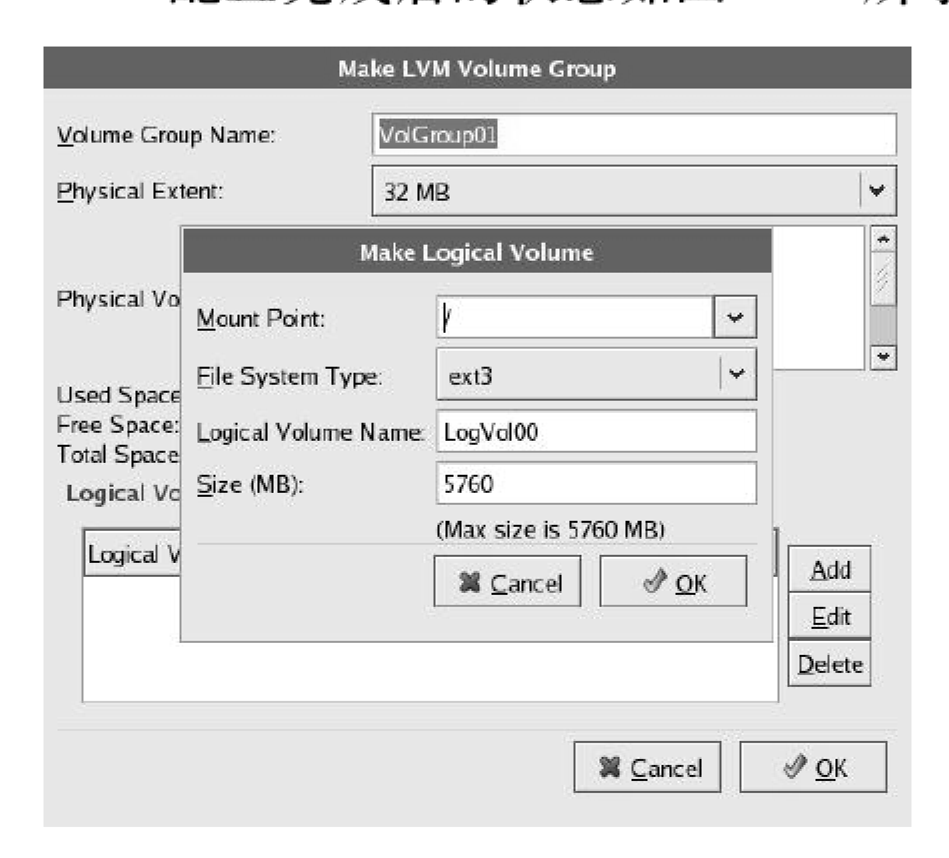


图5.72 创建 LV 并挂载

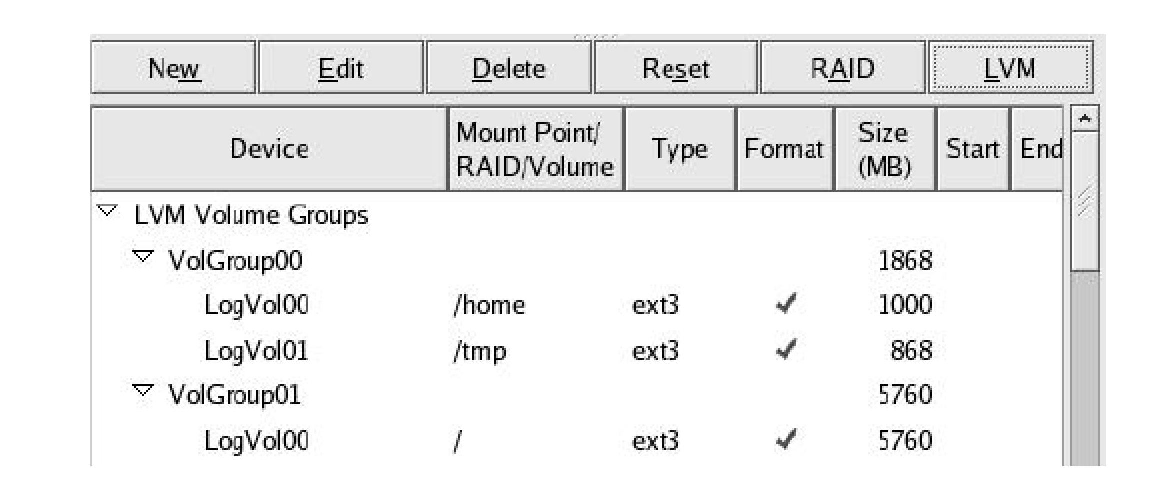


图5.73 配置完成后的状态

5.6.4 卷管理软件的实现

说到这，别以为 LVM 就只会像疯子一样，拿来面团就揉到一起，掰来掰去，什么都不

管，那样岂不真成了马大哈了。它是需要在心里暗自记录的，比如某块物理盘的名称和容量。表面上是和其他物理盘融合到一起，但还是要记住谁是谁，从哪里到哪里属于这块盘，从哪里到哪里属于那块盘，地址多少，等等。

这些信息记录在磁盘上的某个区域，LVM 中这个区域叫做 VGDA。LVM 就是通过读取每块物理磁盘上的这个区域来获取 LVM 的配置信息，比如 PP 大小、初始偏移、PV 的数量和信息、排列顺序及映射关系等。LVM 初始化的时候会读取这些信息，然后在缓存中生成对应的映射公式，从而完成 LV 的挂载。挂载之后，就可以接受 IO 了。比如上层访问某个 LV 的 LBA 0xFF 地址，那么 LVM 就需要通过缓存中的映射关系判断这个地址对应到实际物理磁盘是哪个或哪几个实际地址。假设这个地址实际对应了磁盘 a 的 LBA 0xAA 地址，那么就会通过磁盘控制器驱动直接给这个地址发数据，而这个地址被 RAID 控制器收到后，可能还要做一次转换。因为 OS 层的“物理磁盘”可能对应真正的存储总线上的多块物理磁盘，这个映射就需要 RAID 控制器来做了，原理都是一样的。

卷管理软件对待由 RAID 卡提交的逻辑盘(OS 识别成物理磁盘)和切切实实的物理盘的方法是一模一样的。也就是说，不管最底层到底是单物理盘，还是由 RAID 控制器提交的逻辑物理盘，只要 OS 认成它是一块物理磁盘，那么卷管理器就可以对它进行卷管理。只不过对于 RAID 提交的逻辑盘，最终还是要通过 RAID 控制器来和最底层的物理磁盘打交道。

Linux 下的 LVM 甚至可以对物理磁盘上的一个分区进行卷管理，将这个分区做成一个 PV。

卷管理软件就是运行在 OS 操作系统磁盘控制器驱动程序之上的一层软件程序，它的作用就是实现 RAID 卡硬件管理磁盘空间所实现不了的灵活功能，比如随时扩容。

为什么卷管理软件就可以随时在线扩容，灵活性这么强呢？首先我们要熟悉一个知识，也就是 OS 会自带一个卷管理软件层，这个卷管理软件非常简单，它只能管理单个磁盘，而不能将它们组合虚拟成卷，不具有高级卷管理软件的一些灵活功能。OS 自带的一些简单 VM(卷管理)软件，只会调用总线驱动(一种监视 IO 总线 Plug And Play，即 PNP，即插即用)，发现硬件之后再挂接对应这个硬件的驱动，然后查询出这个硬件的信息，其中就包括容量，所以我们才会在磁盘管理器中看到一块块的磁盘设备。即从底层向上依次是物理磁盘、磁盘控制器、IO 总线、总线驱动、磁盘控制器驱动、卷管理软件程序、OS 磁盘管理器中看见的磁盘设备。

而高级卷管理软件是将原本 OS 自带的简陋的卷管理功能进行了扩展，比如可以对多个磁盘进行组合、再分等。不管是 OS 单一 VM 还是高级 VM，磁盘在 VM 这一层处理之后，应该称为卷比较恰当，就算卷只由一块磁盘抽象而成，也不应该再称作磁盘了。因为磁盘这个概念只有对磁盘控制器来说才有意义。

磁盘控制器看待磁盘，真的就是由盘片和磁头组成。而卷管理软件看待磁盘，会认为它是一个线性存储的大仓库，而不管这个仓库用的是什么存储方式，仓库每个房间都有一个地址(LBA 逻辑块地址)，VM 必须知道这些地址一共有多少。它让库管员(磁盘控制器驱动软件)从某一段地址(LBA 地址段)存取某些货物(数据)，那么库管员就得立即操控他的机器(磁盘控制器)来到各个房间存取货物(数据)。这就是 VM 的作用。

在底层磁盘扩容之后，磁盘控制器驱动程序会和 VM 打个招呼，我已经增大了多少容量了，你看着办吧。卷说：“好，你不用管了，专心在那干活吧，我告诉你读写哪个 LBA 地

址的数据你就照我的话办。”这样之后，VM 就会直接将等待扩容的卷的容量立即扩大，放入池中备用，对上层应用没有丝毫影响。所以 VM 可以屏蔽底层的变化。

至于扩容和收缩逻辑卷，对 VM 来说是小事一桩。但是对于其上的文件系统来说，处理起来就复杂了。所以扩大和收缩卷，需要其上的文件系统来配合，才能不影响应用系统。

5.6.5 低级 VM 和高级 VM

1. MBR 和 VGDA

分区管理可以看作是一种最简单的卷管理方式，它比 LVM 等要低级。分区就是将一块磁盘抽象成一个仓库，然后将这个仓库划分成具体的一库区、二库区等。因为一个仓库太大的话，对用户来说很不方便。比如一块 100GB 的磁盘，如果只分一个区，就显得很不便于管理。有两种方法解决这个问题。

- 可以用低级 VM 管理软件，比如 Windows 自带的磁盘管理器，对这个磁盘进行分区。
- 用高级 VM 管理软件，将这个盘做成卷，然后灵活地进行划分逻辑卷。

这两种方法可以达到将一个仓库逻辑划分成多个仓库的效果。所不同的是分区管理这种低级卷管理方式，只能针对单个磁盘进行再划分，而不能将磁盘合并再划分。

> 思考 对于低级 VM 的分区管理来说，必须有一个东西来记录分区信息，如第一仓库区是整个仓库的哪些房间，从第几个房间开始到第几个结束是第二仓库区，等等这些信息。这样，每次 OS 启动的时候，VM 通过读取这些信息就可以判断这个仓库一共有几个逻辑区域，从而在“我的电脑”中显示出逻辑磁盘列表。那么怎么保存这个分区信息呢？

毫无疑问，它不能保存在内存里，更不能保存在 CPU 里，它只能保存在磁盘上。分区信息被保存在分区表中，分区表位于磁盘 0 磁道 0 磁头的 0 号扇区上，也就是 LBA1 这个地址的扇区上。这个扇区又叫做 MBR，即主引导记录。MBR 扇区不仅仅保存分区表，它还保存了 BIOS 跳转时所需要执行的第一句指令代码，所以才叫做主引导记录。

BIOS 代码都是固定的，它每次必定要执行 LBA1 扇区上的代码。如果修改 BIOS，让它执行 LBA100 扇区的代码，也可以，完全可以。但是现在的 BIOS 都是执行 LBA1 处的代码，没人去改变。而新出的规范 EFI 将要取代 BIOS，并且在安腾机上已经使用了，一些苹果笔记本也开始使用 EFI 作为 BIOS 的替代。在 EFI 中可以灵活定制这些选项，比如从哪里启动，不仅可以选择设备，还可以选择设备上的具体地址。

MBR 中除了包含启动指令代码，还包含分区表。通常启动时，程序都会跳转到活动分区去读取代码做 OS 的启动，所以必须有一个活动分区。这在分区工具中可以设置。

高级卷管理软件在划分了逻辑卷之后，一定要记录逻辑卷是怎么划分的，比如 LVM 就需要记录 PV 的数量和信息、PP 的大小、起始位置及 LV 的数量和信息等。这些信息都要保存在磁盘上，所以也要有一个数据结构来存储。这个数据结构，LVM 使用 VGDA(Volume Group Descriptor Area)。每次启动系统，VM 就是通过读取这些数据来判断目前的卷情况并挂载 LV 的。VGDA 大致结构示意图如图 5.74 所示。

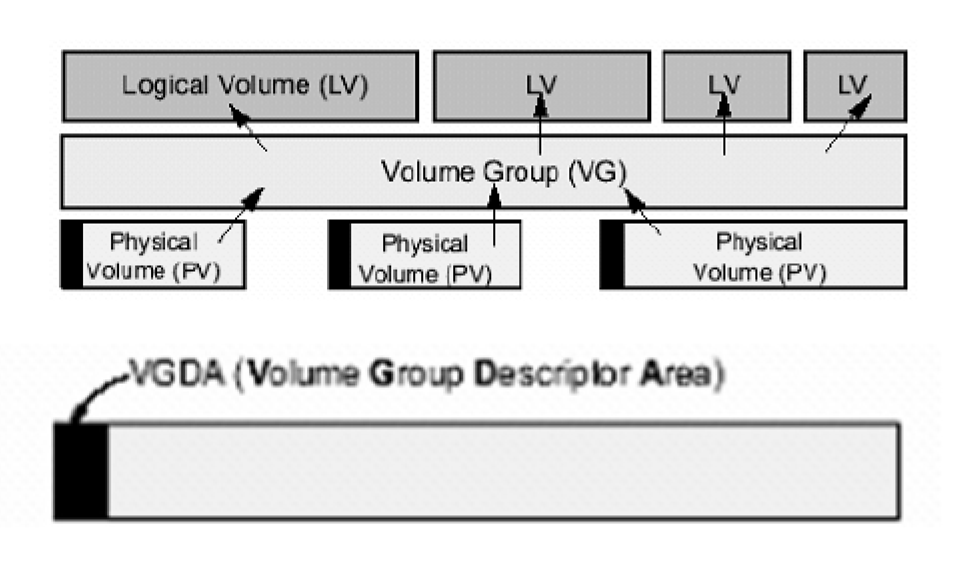


图5.74 VGDA 示意图

不管是 MBR 中的分区表，还是 VGDA9 数据结构，一旦这些信息丢失，逻辑卷信息就会丢失，整个系统的数据就不能被访问。

低级 VM 在给磁盘分区的时候，会更新 MBR 中的分区表；高级 VM 做逻辑卷的时候，同样也会更新 VGDA 中的数据。其实高级 VM 初始化一组新磁盘的时候，并没有抛弃 MBR。因为它们除了写入 VGDA 信息之外，也要更新 MBR 扇区中的分区表，将用于启动基本操作系统的代码单独存放到一个小分区中，并标明分区类型为 bootable 类型，证明这个分区是用于在卷管理模块还没有加载之前启动操作系统的。并将磁盘所有剩余容量划分到一个分区中，并标明这个分区的类型，如 AIX 类型。

在安装 Linux 的时候，必须单独划分一个/boot 分区，这个分区就是用于启动基本操作系统用的，100MB 大小足矣。启动操作系统所必需的代码都放在这个分区。同样 AIX 系统也要保留一个分区用来启动最基本的操作系统代码。这也是 AIX 在进行了 Mirrorvg 镜像操作之后，需要执行 BOSboot 命令来写入 boot 分区的内容的原因，因为 boot 分区没有参与 VM 管理。这个启动分区是不能做到 VM 中的，因为 VM 代码不是在 BIOS 将控制交给 OS 的时候一开始就执行的。

总之，高级 VM 没有抛弃 MBR 分区的解决方案，而是在 MBR 基础上，又增加了类似 VGDA 这种更加灵活的数据结构来动态管理磁盘。

2. RAID 功能

高级 VM 软件一般均带有软 RAID 功能，可以实现逻辑卷之间的镜像。更有甚者，有些 VM 甚至实现了类似 RAID 0 的条带化。在卷的级别条带化，达到在物理盘级别条带化同样的目的。但是如果磁盘已经被硬件 RAID 控制器条带化过了，并且这些 LUN 是在一个 RAID Group 中，那么 VM 再来条带化一下子不但没有必要，而且可能二次条带化会将效果抵消。

Windows 的动态磁盘 VM 还可以以纯软件方式实现 RAID 5，所有计算都靠 CPU，所以也就注定了它比硬件 RAID 更灵活，但在高系统负载的情况下，它相比硬件 RAID 来说速度和性能稍差。

5.6.6 VxVM 卷管理软件配置简介

VxVM 是 Veritas 公司开发的一个高级卷管理软件，支持 RAID 0、RAID 1、RAID 01 和 RAID 5 四种软 RAID 模式，支持动态扩大和缩小卷容量。

下面的例子是在一个 UNIX 系统中对 4 块磁盘做卷管理的案例。所有命令均在 UNIX 的 Shell 下执行。

1. 创建磁盘组

磁盘组就是将所有磁盘作为一个大的资源池，卷将在这个池中产生。

1】 首先，初始化硬盘。

```
vxdisksetup -i disk1
vxdisksetup -i disk2
vxdisksetup -i disk3
vxdisksetup -i disk4
```

2】 然后创建一个名为“DataDG”的磁盘组，该磁盘组包含了 disk1、disk2、disk3 和 disk4 四个磁盘。

```
vxdg init DataDG disk1 disk2 disk3 disk4
```

除了这种方法，用户还可以用以下方法来创建磁盘组。

```
vxdg init DataDG DataDG01=disk1; (创建一个只包含 disk1 的磁盘组);
    vxdg -g DataDG adddisk DataDG02=disk2 (将 disk2 加入到该磁盘组);
    vxdg -g DataDG adddisk DataDG03=disk3 (将 disk3 加入到该磁盘组);
  vxdg -g DataDG adddisk DataDG03=disk4 (将 disk4 加入到该磁盘组);
```

如果用户在所需磁盘空间不足，需要扩容的时候，利用添加磁盘到磁盘组的方法，就可以在不破坏现有环境的情况下扩大系统的容量。

2. 创建卷

1】 创建卷必须指明在哪个磁盘组下面创建，最常用的方法如下。

```
vxassist -g DataDG make DataVolA 5g
```

该命令将在 DataDG 磁盘组上创建名为“DataVolA”的卷，卷的大小是 5GB。

2】 如果用户希望该卷只创建在 disk1 和 disk2 上面，不占用 disk3 和 disk4 的空间，那么可以执行下列命令。

```
vxassist -g DataDG make DataVolA 5g disk1 disk2
```

3】 创建一个 5GB 大小的条带卷(RAID 0)。

```
vxassist -g DataDG make DataVolB 5g layout=stripe
```

这样就在 DataDG 磁盘组上面建立了一个名为“DataVolB”的 5GB 大小的条带卷。



4 块物理磁盘中，只有 5GB 的空间是条带化的，剩余的空间还是常规的磁盘空间。为什么呢？条带化 RAID 0 不是需要至少两块物理硬盘么？这就是卷管理软件的优越性了。我们上文提过，卷管理软件将物理磁盘划分为 PP 和 LV，所以有了更加细粒度的存储单位，条带化可以在这些 LV 之间进行，而其他 LV 不受影响。

4】 创建 RAID 5 格式的卷。

```
vxassist -g DataDG make DataVolC 5g layout=RAID 5
```

> **注意** RAID 5 至少需要 3 块盘，否则不能成功。因为两块盘的 RAID 5，还不如做 RAID 1。但是 3 块盘的 RAID 5 不能获得并发 IO 性能。

5） 创建镜像卷(RAID 1)。

```
vxassist －g DataDG make DataVolD 5g layout=mirror
```

6） 创建 RAID 10 卷。

```
vxdg  init  RAID 10dg disk1 disk2 disk3 disk4  创建磁盘组
  vxassist －g RAID 10dg  RAID 10vol 5g  layout=mirror-stripe
```

7） 创建 RAID 01 卷。

```
  vxdg  init  RAID 01dg  disk1 disk2 disk3 disk4  创建磁盘组
 vxassist －g RAID 01dg  RAID 01vol  5g  layout= stripe-mirror
```

3. 创建文件系统并使用

```
mkfs －F vxfs /dev/vx/rdsk/DataDG/DataVolA;
mount －F vxfs /dev/vx/dsk/DataDG/DataVolA /mnt
```

以上例子将卷 DataVolA 格式化成 VxFS(Veritas 公司的文件系统)格式，然后挂载于/mnt 目录下，执行命令 cd /mnt 之后，就可以读写这个卷的内容了。

4. 动态扩大和缩小卷

1） 将卷空间增加到 10GB。

```
vxassist -g DataDG growto DataVolA 10G
```

2） 更改之后，卷的容量将会变成 10GB。或者用 vxresize 命令。

```
vxresize -g DataDG DatavolA 10G
```

3） 将卷容量增加 10GB。

```
vxassist -g DataDG growby DataVolA 10G
```

4） 或者用 vxresize 命令：

```
vxresize -g DataDG DatavolA +10G
```

这样，更新之后卷的容量将在原来的基础上增加 10GB 大小。

5. 文件系统动态扩容

卷扩容之后，只是在卷的末尾增加了一块多余空间。这块空间如果没有文件系统的管理就无法存放文件，所以必须让文件系统将这块多余的空间利用起来。

```
fsadm -F vxfs -b 10240000 －r dev/vx/rdsk/DataDG/DataVolA /mnt
```

6. 文件系统缩小

如果决定将某个卷缩小以省出更多空间，则在缩小卷空间之前，必须缩小文件系统的空间。也就是说，被裁掉的卷空间上存放的数据，需要转移到卷剩余的空间上存放，所以

剩余空间必须足够，以便容纳被裁掉空间中的数据。

```
fsadm -F vxfs -b 5120000 -r dev/vx/rdsk/DataDG/DataVolA /mnt
```

以上命令将这个卷上的文件系统缩小至 5GB 大小。剩余的 5GB 没有数据，可以被裁剪掉。

7. 卷容量缩小

在缩小了文件系统之后，卷容量方可缩小。

```
vxassist -g DataDG shrinkto DataVolA 5G
vxresize -g DataDG DataVolA 5G
```

上面的两个命令均可以使 DataVolA 卷的容量变为 5GB。

```
vxassist -g DataDG shrinkby DataVolA 5G
vxresize -g DataDG DataVolA -5G
```

上面的两个命令均可以使 DataVolA 卷的容量在原来的基础上缩减 5GB。

8. 从磁盘组中移除磁盘

若想从磁盘组中移除一块或者几块物理磁盘，则必须先将待移除物理磁盘上的数据转移到磁盘组中的其他物理磁盘的剩余空间中，这个动作通过下面的命令完成。

```
vxevac -g DataDG DataDG04 DataDG03
```

上面的命令将 disk4 中的数据转移到 disk3 上。除了容量改变之外，不会影响卷的其他信息。

```
Vxdg -g DataDG rmdisk disk4
```

上面的命令将已经没有数据的 disk4 物理磁盘从磁盘组 DataDG 中移除(逻辑移除)。

```
vxdiskunsetup -C Disk4
```

上面的命令将 disk4 物理磁盘从整个 VxVM 管理模块中注销。

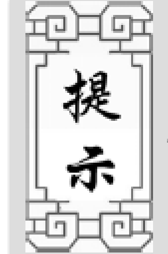
5.7 大话文件系统

5.7.1 成何体统——没有规矩的仓库

话说这一天，老道闲来无事，在后山溜达。他走到了武当的粮库门口，发现这里堵了一大帮人。老道上前一问，原来这些人都是各个院来领取粮食的。只见他们一拥而上，进入仓库就各自找自己的房间去搬粮食。老道一看，怎么这么乱呢？就不能有个顺序么？

他向其中一个小道打听了一下，这才知道，造成这种乱七八糟进入粮库搬粮食局面的原因，是因为当初没有好好规划仓库。上个月，各个院从山下各自运了粮食上来，当时的政策是大家各自进入仓库，自己找房间放自己的粮食，自己找了哪些房间放粮食，自己记住了。到取粮食的时候，大家根据自己记录的房间来进入取粮。这个政策看似没什么可非议的，实则不然。如今山下粮食供应紧张，造成大家各顾各的，没有顺序，岂能不乱？老道进入粮仓一看，眼前一片狼藉！土豆、西红柿洒落得满地都是。这间房放这样，那间房

放那样，就不能顺序地堆放粮食蔬菜？成何体统！！


在早期的计算机系统中，每个程序都必须自己管理磁盘，在磁盘中放自己的数据，程序需要直接和磁盘控制器打交道。有多少个程序要利用磁盘，就有多少个和磁盘交互的驱动接口。

老道摇了摇头，得想个办法彻底解决这个问题。老道回到了书房，闭目思索。首先大家不能都堵在门口，那么必须让他们排起队来。其次，每个人各顾各，自己记录自己用了哪间房子，一个是浪费，另一个是容易造成冲突。一旦某个人记错了，就会影响其他人。那么就应该只让一个人记录所有人的信息，他自己不会和自己冲突。同样这个人也要充当一个门卫的作用，接待来取粮或者送粮的人，让他们按一定的顺序来运作。

最终决定就应该是这样的：找一个人，这个人的职责就是接待来取粮或者送粮的人，把要取的或者要送的粮食的名称和数量等信息先登记在这个人的一个本子上，然后由这个人来合理地选择仓库中的房间，存放或提取登记在案的粮食，而且提取或放入粮食之后要将本子上的记录更新，以便下次备查。嗯，这么做就好多了，哈哈哈哈！这天晚上的北斗七星，光芒格外耀眼。

5.7.2 慧眼识人——交给下一代去设计

第二天，老道亲自挑选了一位才思敏捷、内向稳重、善于思考的道士来担任这个重要的角色。让他和库管员一起完成管理粮库的工作，给他起了一个职称，叫做理货员。并且将自己的想法告诉了这位道士，让他当晚就考虑出一套符合这个思想的方法，还可以做出改进意见。

就这样，又过了一晚。第三天，这位道士上任了。一大早，张老道就在暗中观察。这时候，一个送粮食的人来了，他带了 1024 斤土豆和 512 斤白菜。这人还是按老习惯，上来就往仓库闯。小道士截住了他：“道长且慢！您需要送的是什么蔬菜？”那人道：“土豆和白菜！”小道士又道：“土豆多少斤？”答曰：“土豆 1024 斤。”(上面这个过程就是应用程序和 FS 的 API 交互的过程)。小道士笑道：“道长尽可放心将土豆交于我，我自当为您找房间存放。”然后小道士到仓库中找了 2 个空房间，每个房间放了 512 斤土豆。并在本子上记录：“土豆 1024 斤房间 1－2。”接着他就命令库管员来搬运货物到相应的房间。

道士给每个库区都预备了一个记录本。小道士不关心具体房间到底在仓库哪里，怎么走才能达到，这些事情统统由库管员来协调。小道士同样也不关心来送货物的人到底送的是什么货物。如果送粮的人告诉他，请给我存放 rubbish 1000 斤，道士眼都不眨照样给他存放。一旦仓库的房间都满了，小道长再次命令库管员搬运货物的时候，库管员就会告诉他，已经没有房间了。那么道长就告诉来存放货物的人：“对不起，空间不足”。

用同样的方法，小道士将那人的白菜，也放到了一间房中，记录下：“白菜 512 斤 房间 3”。然后向那人说到：“这位道长，您下次来取的时候，直接向我说要某厨房存放的土豆多少斤就可以了，我会帮您找到并取出。”那人非常满意地离去了。接着又有很多人也来送取冬瓜、南瓜、西瓜、大米、面粉等粮食蔬菜，小道士一一对应，有条有理。小道士也专门给自己在每个库区中预留了几间房，用于存放他那一本本厚厚的记录。老道一旁看

了，频频点头，“嗯，前途无量，前途无量啊，啊哈哈哈哈哈！！！”

过了几天，张真人又来探查。此时只见有个人一下送来 10000 斤大米。小道长开始只是表示吃惊，并没有多想，仍旧按照老办法，记录“大米 10000 斤，房间 4－4096”。接着又来了一位要存放 65535 斤小麦的。这下可苦了小道士了，把他累得够呛。随着全国粮食大丰收，存粮数量动辄上万斤。这让小道士苦不堪言，他决定思考一种解决方法。第二天，小道长将每 8 个房间划分为一个逻辑房间，称作“簇”。第一簇对应房间 1、2、3、4、5、6、7、8，第二簇对应房间 9、10、11、12、13、14、15、16。依此类推。这样道士记录的数字量就是原来的八分之一了。比如 4000 斤粮食，只需记录“簇 1”就可以了。老道心中暗想，“嗯，不错，我没看错人！！”这一年，因为大丰收，粮食降价了。农民丰产不丰收，很多农民打算第二年不种粮了，改做其他小生意。

5.7.3 无孔不入——不浪费一点空间

第二年，果然不出张老道所料，全国粮食大减产，价格飞涨，全面进入恐慌阶段。张真人悬壶济世，开仓放粮，平息物价。这一举动受到了老百姓的称赞和感激，但也招致了一小部分奸商的忌恨。

放粮消息宣布之后，山下老百姓都排队来武当买粮。这可忙坏了理货员道士，连续几天没休息，给老百姓取粮食。一个月之后，武当粮库存粮已经所剩无几，张老道和众院道士每天省吃俭用，为的是给老百姓多留点存粮。

大恐慌的一年，终于熬过去了。农民一看粮食价格那么高，第三年又都准备种粮了。不出意料，这一年粮食又得丰收！张老道提前考虑他的粮库在这一年的使用问题了，他叫来理货员道士，让他回去考虑一个问题：经过了去年的折腾，仓库中的存货是零零散散，乱七八糟，为了准备这一年大量粮食涌入仓库，必须解决这个问题，让他回去考虑解决办法。其实张老道早就在心里盘算出了解决办法了。

第二天，理货员趁人少的时候，就命令库管员：“请帮我把房间 xxxxx 的货物移动到房间 xxxx 处，请帮我把房间 xxx 货物移动到房间 xxx 处……”。

这可累坏了库管员。但是经过几个时辰的整理之后，仓库里的货物重新变得连续，井井有条。老道称赞说：“不错！继续努力！”。

这天晚上，小道长也没闲着，他继续思考，今天是有时间整理货物，如果一旦遇到忙的时候，没有时间整理货物，那麻烦就大了，得想一种一劳永逸的办法。有些人来送完粮食之后，第二天就来取了，这个真是头疼了。因为我都按照顺序将每个人的粮食连续存放到各个簇中，他一下取走了，对应的簇就空了。如果再有人来，他带的货物数量如果这个空簇能存下还好，可以接着用。如果存不下呢？还得找新的连续空簇来存放。如果这种情况出现太多，那么整个仓库就是千疮百孔，大的放不下，小的放下了又浪费空间。……真头疼。他冥思苦想，最后终于想出一个办法。

一早仓库还没有开门的时候，小道长就来了，他把所有记录本都拿了出来，进行修改。他原本对每个来送货的人，都只用一条简单记录来描述它，描述中包含 3 个字段：名称、大小和存放位置。比如冬瓜 10000 斤 簇 1－3。此时仓库中，虽然总空余空间远远大于 10000 斤的量，但是已经没有能连续地放入 10000 斤大小的簇空间，那么这个货物就不能

被放入仓库，而这是不能容忍的一种浪费。有一个办法，就是上面说过的，找空闲时间来整理仓库，整理出连续的空间来。这次小道长想出了另一个方法，就是将货物分开存放，并不一定非要连续存放在仓库。因为仓库已经被逻辑分割成以簇(4 个房间)为最小单位存放货物。那么就可以存在类似这样的描述方式：冬瓜 10000 斤 簇 2、6、19。也就是说这 10000 斤的冬瓜是分别被按顺序存放在仓库的 2 号簇、6 号簇和 19 号簇中的。取出的时候，需要先去 2 号取出货物，再跨过 3 个簇去 6 号，再跨过 13 个簇去 19 号。都取出后再交给提货人。这样确实慢了点，但是完美地解决了空间浪费的问题。

5.7.4 一箭双雕——一张图解决两个难题

粮食大丰收果然又被张老道猜中了，这次小道长是应对自如，一丝不乱。老道啧啧称赞！但是老道却从小道士的记录中，又看出了一些问题，他告诉小道士，要继续思考更好的解决办法。小道士心很灵，他知道这个方法确实解决了问题，但是有缺陷，会有后患，只不过现在的环境并没有显示出来。这天晚上，小道在仓库睡觉，没有回去。

提示

看着他那些记录，只见上面一条一条，一行一行的，却也比较有条理。但是仔细一看发现，每一条记录的最后一个字段，也就是描述货物存放在哪些簇的那个字段，非常凌乱，因为每个人送来的货物数量不一样，那么就注定这个字段长短不一，显得非常乱。

现在记录不是很多，记录一旦增多，每次查询的时候就很不好办。而且要找一个未被占用的簇，需要把所有已经被占用的全找出来，然后才去选择一个未被占用的簇，分配给新的货物存放。这个过程是非常耗时间的，货物少了还可以，货物一多，那可就费劲了。“嗯，张真人让我继续思考，确实是有道理的，这两个隐患，确实是致命的，尤其是第二个。得继续找新方法”。

小道士继续思考。第一个问题，要想解决长短不一的毛病，最简单的就是给他一个定长的描述字，这仿佛是不可能的，有的需要 1 个簇就够了，有的却需要 10 个甚至 100 个，如果把这需要 10 个簇的和需要 1 个簇的，都用 1 个簇来描述，那么确实非常漂亮了，记录会非常工整。

想到这里，小道士累了，想出去走走。他溜达到一个路口，看见路口上有路标牌，上面写着：“去会客厅请走左边，去习武观请走右边。”小道士顺着路标指向，走了右边，然后又遇到一个路标：“去习武观请走左边，下山请走右边。”道士走了左边，最终来到了习武观。他看着习武观正中央的那个醒目的“道”字，忽然眼前一亮！

他迅速原路返回到粮库，拿出记录本，将其中一条记录改为：“冬瓜 10000 斤 首簇 1”。每条记录都改成这种形式，也就是只描述这个货物占用的第一个簇的号码，这样完美解决了记录长短不一的问题，那么后续的簇呢？只知道首簇，剩余的不知道，一样不能全部把货物取出。

所以小道士参照路标的形式，既然知道了首簇号，那么如果找到首簇，再在首簇处作一个标记，写明下一个簇是多少号，然后找到下一个簇取货，然后再参照这个簇处的路标，到下下个簇处接着取货，依此类推，如果本簇就是这批货物的最后一簇，那么就标识：“结

束，无下一簇”。比如：“冬瓜 10000 斤 首簇 1”这个例子，先把 4096 斤冬瓜放到簇 1 中，然后在簇 1 的门上贴上一个标签：“簇 10”，这就表明下一簇是 10 号簇。继续向 10 号簇中存入 4096 斤冬瓜，此时还剩 808 斤冬瓜没放入，还需要一个路标，于是在 10 号簇的门上再贴一个标签：“90 号”。然后去 90 号簇放入剩下的 808 斤冬瓜。

第二天，张老道继续来视察。老道一看他的记录，不由地一惊！“一个晚上就想到了这种绝妙方法。嗯，此人大有前途！”老道频频点头称赞。然后老道进仓库查看，一看有些簇的门上，贴着标签，老道立即明白了小道长的做法，和小道说：“孩子，不错，但是还需要再改进！”

小道心里盘算，“嗯，这个方法是解决了第一个问题，但是每个簇门上都贴一个标签，这样是不太像样。而且寻找未被占用的簇的效率还是那么低，还是需要把所有已经占用的簇找出来，再比对选出没有使用的空簇。而且我这么一弄，找空簇的效率比原来还差了，因为原来已经使用的簇都会被记录在货物描述中的字段中，现在把这个字段缩减成一个字了，这样每次找寻的时候，还得去仓库中实际一个门一个门地去抄下已经使用的簇，还不如直接在本子上找来的快。这个问题得解决！”

思考

既然要拿掉贴在门上的标签，那么就必须找另外一个地方存放标签，所以只能存放到我的记录本上。可是各个簇的路标我都记录在本子上，用一个什么数据结构好呢？货物描述那三个字段肯定不能再修改了，那样已经很完美了，不能破坏它。那么就需要再自己定义一个结构来存放这些路标之间的关系，而且每个货物的路标还不能混淆，混了就惨了。他在纸上写写画画，不知不觉把整个仓库的簇画出来了，从第一个簇，到最后一个簇，都用一个方格标识，然后他参照“冬瓜，10000 斤，首簇 1”这个例子，下一簇是簇 10，那么他在簇 1 的格子上写上了“簇 10”，然后他找到第 10 个格子，也就是代表簇 10 的格子，在簇 10 格子里面写上“簇 90”，也就是 10 号簇的下一簇路标。然后继续找到 90 号簇，此时他在这个格子里写上“结束”。接着他又举了几个例子，分别画了上去。就这么逐渐睡着了。

第二天早晨，小道士迷迷糊糊地起来了，只见张道长已经在他的面前，带着赞许的笑容。“孩子，你累了，不错不错，你终于把所有问题都解决了啊！”张老道摸着小道士的脑袋，称赞地说道。小道士还不知道是怎么回事呢，他告诉张老道说，他还没想出来呢。老道大笑说：“哈哈哈哈，你看看你画在纸上的图，这不是已经解决了么？哈哈哈哈哈。”说完老道扬长而去。

小道士一头雾水，看着那张画，这才想起了昨晚的思考。“对啊，这张图不就行了么？这就是我所要找的数据结构啊！”接着，小道士把图重新画了一张，工工整整地夹在了记录本里面。这时，来了一个取货的人，他告诉小道士说：“二库区，南瓜，10000 斤”。道士说：“稍等”。然后立即查询二库区的记录本，找到南瓜的记录，发现首簇是 128。然后立即到那张图上找到第 128 号簇所在的格子，发现上面写的是“簇 168”。继续找到第 168 号格子，上面写的是“簇 2006”。立即找到第 2006 个格子，只见上面写的是“结束”。然后他通知库管员“请帮我将第 128、168、2006 三个簇的货物提取出来给我”。不一会

儿，货物到了，交货签字。小道士恍然大悟，“太完美了！！”

紧接着，又来了一个存货的人，他有西瓜 500 斤要存放到 1 库区。小道士立即查看那张图，一目了然。只要格子上没有写字的就是空簇，就可以用来存放货物。所以道士立即找到一个空着的 50 号簇来存放这 500 斤西瓜。存放完毕之后，在对应的这个格子上写上“结束”，因为 500 斤的数量一个房间就够了，更不用说一个簇了(最多 8 个房间)。接着也在 1 库区的记录本上增加一条记录“西瓜　500 斤　首簇 50”。

道士发现，第二个问题也就是查找未被使用的簇的问题，自从有了这张图，就自然解决了。道士非常兴奋，同时也佩服张真人，是他引导着自己一步一步解决问题的。

5.7.5　宽容似海——设计也要像心胸一样宽

随着仓库业务的不断成熟，小道士的技能越来越熟练，他开始考虑描述货物的三个字段：名称、数量、存放的第一个簇。随着国民生产力水平不断提高，各种层出不穷的产品被生产出来，它们有些具有一些奇特的属性。所以小道士准备增加字段来表述一件货物更多的属性，比如送货时间、只读、隐藏等各种花哨属性。同时，那张图也不能满足要求了，因为随着生产力发展，仓库每平方米造价越来越低，武当决定扩大仓库容量。这样仓库中所包含的簇数量就大大增加了，甚至成几何数量级增长，所以簇号码越来越大，甚至超过了亿。要记录这么多位的数字，本来那个小格子就写不开了，所以需要增大格子的宽度，以便能写下更多的数字位数。以前每个格子是 2 字节(16 位)长度，现在扩展到了 4 字节(32 位)。而据传江湖上另一位大侠已经将格子的宽度扩展到了 128 位。

要问这位小道长姓字名谁？因为当时张真人收留他的时候，发现他身板有点软，不适合练武。但思维敏捷，适合练心法，所以给他一个道号叫做微软。

就这样，仓库又运作了 2 年。

5.7.6　老将出马——权威发布

仓库存储容量不断增加，仓库管理技术方面却并没有什么进步，还是沿袭 2 年前那一套运作模式。这显然已经不适应现代仓库了，所以造成入库等待、处理速度逐渐变慢等一系列的问题。张真人决定跟上时代，要研究出一套新的仓库运作模式，并且定义出一个规范，让全天下的仓库都沿袭这个规范来运作。张老道先仔细考察了微软道士的运作模式，然后根据现代仓库管理的特点，提出了一系列的解决方案。

现代仓库管理要求入库出库速度快，由于在仓库硬件方面提高很快，有了更加新式的传送带和机器人等机器，所以大大提高了操作简化度，减轻了库管员的负担。库管员只需要阅读机器的随机手册(驱动程序)便可以轻松地完成操作。与此同时，对于理货员这块技术并没有什么新的突破，因为理货这块主要靠好的算法，并不需要硬件支持，除了那些记录本之外。而从仓库中取出记录本的速度，由于库管员操作迅速，所以也不在话下。关键就看理货算法了。这是任何硬件都不能解决的问题。

首先张老道通过观察、记录，发现一般货物就算是存放到不连续的簇中，这些簇往往也是局部连续的，比如 1、2、3、5、6、7、100、101、102，其中 1、2、3 就是局部连续，5、6、7 也是，100、101、102 也是。而不太可能出现一个货物占用了 1、56、168、2008

簇这种情况。如果此时不是一个簇一个簇地去找路标，而是一段一段地去找，这样会节约很多时间和精力。比如簇段 1－3，簇段 5－7，簇段 100－102。这样就大大简化了路标。还有其他的一些改进方式，如直接将一些小货物存放到它们描述记录中(驻留文件)。只有描述记录中放不下时，才到仓库其他区域找一些簇来存放，然后记录这些簇段。

微软道士将他的记录本上的信息，称为 Metadata，元数据。也就是用来描述其他数据是怎么组织存放的一种数据。如果记录本丢失，那么纵然仓库中货物完好无损，也无法取出。因为已经不知道货物的组织结构了。

张真人最后把微软道士实现的一共三种仓库运作管理模式，分别叫做 FAT16、FAT32 和 NTFS，并取名为小道藏龙。

5.7.7　一统江湖——所有操作系统都在用

后来张老道把这套管理模式移植到了磁盘管理上，也就是轰动武林的所谓“文件系统”。对应仓库来说，送货人送来的每一件货物都称作“文件”。取货时，只要告诉理货员文件名称、所要取出的长度及其他一些选项，那么理货员就可以从仓库中取出这些数据。

在一个没有文件系统的计算机上，如果一个程序要向磁盘上存储一些自己的数据，那么这个程序只能自己调用磁盘控制器驱动(无 VM 的情况下)，或者调用 VM 提供的接口，对磁盘写数据。而写完数据后，很有可能被其他程序的数据覆盖掉。引入文件系统之后，各个程序之间都通过文件系统接口访问磁盘，所有被写入的数据都称为一个文件，有着自己的名字，是一个实体。而且其他程序写入的数据，不会将其他人的文件数据覆盖掉，因为文件系统会计算并保障这一点。

除此之外，不仅张真人的 NTFS 文件系统取得了巨大的成功，适应了现代的要求。与此同时，少林的雷牛方丈也创造出了其他的文件系统，比如 EXT 一代、二代、三代和 JFS 等文件系统。一时间文件系统思想的光环是照耀江湖！！

5.8　文件系统中的 IO 方式

我们来看一下，有了文件系统之后，整个系统是个什么架构。

图 5.75 为 Windows 系统的 IO 简化流程图。

图中的 IO Manager 是操作系统内核的一个模块，专门用来管理 IO，并协调文件系统、卷、磁盘驱动程序各个模块之间的运作。整个流程解释如下。

1】 某时刻，某应用程序调用文件系统接口，准备写入某文件从某个字节开始的若干字节。

2】 IO Manager 最终将这个请求发送给文件系统模块。

3】 文件系统将某个文件对应的逻辑偏移映射成卷的 LBA 地址偏移。

4】 文件系统向 IO Manager 请求调用卷管理软件模块的接口。

5】 卷管理软件将卷对应的 LBA 地址偏移翻译映射成实际物理磁盘对应的 LBA 地址偏移，并请求调用磁盘控制器驱动程序。

6】 IO Manager 向磁盘控制器驱动程序请求将对应 LBA 地址段的数据从内存写入某块

物理磁盘。

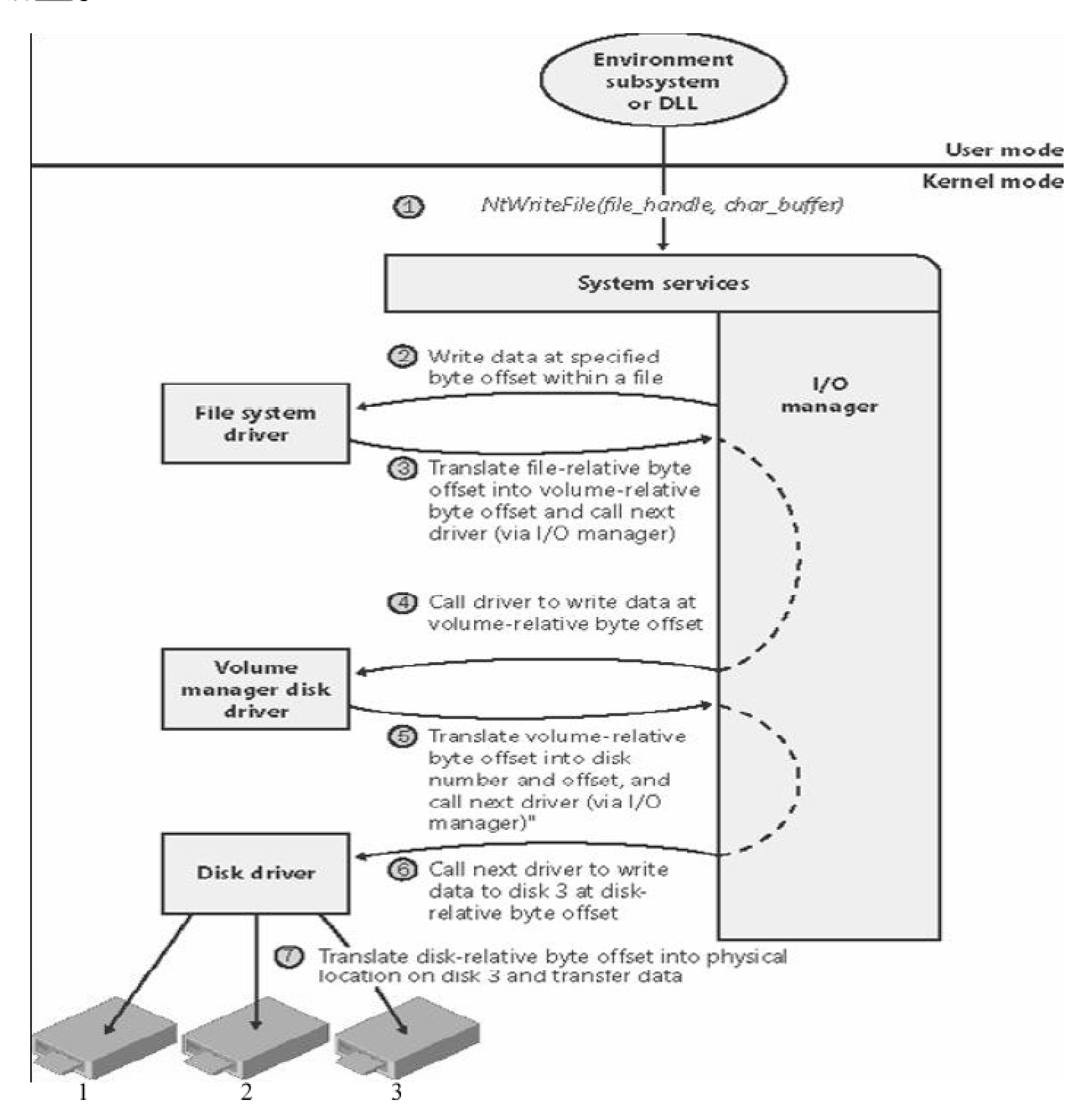


图5.75 Windows 系统的 IO 流程图

文件系统的 IO 包括同步 IO、异步 IO、阻塞/非阻塞 IO 和 Direct IO

同步 IO：同步 IO 是指程序的某一个进程或者线程，如果某时刻调用了同步 IO 接口，则 IO 请求发出后，这个进程或者线程必须等待 IO 路径上的下位程序返回的信号（不管是成功收到数据的信号还是失败的信号），如果不能立刻收到下位的信号。则一直处于等待状态，不继续执行后续的代码，被操作系统挂起，操作系统继续执行其他的进程或者线程。

而如果在这期间，倘若 IO 的下位程序尚未得到上位程序请求的数据，此时 IO 路径上的下位程序又可以选择两种动作方式，第一是如果暂时没有得到上位程序请求的数据，则返回通知通告上位程序数据未收到，而上位程序此时便可以继续执行；第二种动作则是下位程序也等待它自己的下位程序来返回数据，直到数据成功返回，才将数据送给上位程序。前者就是非阻塞 IO，后者就是阻塞 IO 方式。

同步+阻塞 IO 是彻底的堵死状态，这种情况下，除非这个程序是多线程程序，否则程序就此挂死，失去响应。同理，异步+非阻塞的 IO 方式则是最松耦合的 IO 方式。

异步 IO：异步 IO 请求发出后，操作系统会继续执行本线程或者进程中后续的代码，直到时间片到时或者因其他原因被挂起。异步 IO 模式下，应用程序的响应速度不会受 IO 瓶颈的影响，即使这个 IO 很长时间没有完成。虽然应用程序得不到它要的数据，但不会影响其他功能的执行。

基于这个结果，很多数据库在异步 IO 的情况下，都会将负责把缓存 Flush 到磁盘的进程(Oracle 中这个进程为 DBWR 进程)数量设置成比较低的数值，甚至为 1。因为在异步 IO

的情况下，Flush 进程不必挂起以等待 IO 完成，所以即使使用很多的 Flush 进程，也与使用 1 个进程效果差不多。

异步 IO 和非阻塞 IO 的另一个好处是文件系统不必立刻返回数据，所以可以对上层请求的 IO 进行优化排队处理，或者批量向下层请求 IO，这样就大大提升了系统性能。

Direct IO：文件系统都有自己的缓存机制，增加缓存就是为了使性能得到优化。而有些应用程序，比如数据库程序，它们有自己的缓存，IO 在发出之前已经经过自己的缓存算法优化过了，如果请求 IO 到达文件系统之后，又被缓存起来进行额外的优化，就是多此一举了，既浪费了时间，又降低了性能。对于文件系统返回的数据，同样也有这个多余的动作。所以文件系统提供了另外的一种接口，就是 Direct IO 接口。调用这种接口的程序，其 IO 请求、数据请求以及回送的数据将都不被文件系统缓存，而是直接进入应用程序的缓存，这样就提升了性能。此外，在系统路径上任何一处引入缓存，如果是 Write Back 模式，都将带来数据一致性的问题。Direct IO 绕过了文件系统的缓存，所以降低了数据不一致的风险。



大话磁盘阵列

- 磁盘阵列
- SCSI
- LUN
- 前端/后端

两三块磁盘做 RAID 0 或 1，四五块磁盘做个 RAID 3、4、5 是小事一桩，不过太没魄力。要玩就弄个几十块盘，那才过瘾。这不，有人发明了专门装这些磁盘的大柜子，我们这就去看看这柜子是怎么回事儿吧。

退隐江湖——太累了，该歇歇了

自从张真人创立了降龙三掌之后，江湖各门各派争相修炼，商人不断推出基于降龙掌的新商品。江湖上浮躁之气再次袭来，很少有人去钻研底层功夫了，都是拿来就用，不思进取。几十年过去了，张老道已经成了头发苍白的老人。

这天晚上，人少星稀。唯独天上的北斗七星，光芒还是那么灿烂，仿佛已逝去百年的七星大侠，还在天上苦苦钻研？

张真人如今也已经是白发苍苍，可是知己已不在。一百年来，江湖上为了利益你争我抢，反目成仇，打打杀杀。呜呼哉！！！！难道这个江湖真要从此衰败么？张老道失望至极。

闻道

尘世浮华迷人眼，
梦中情境亦非真。
朝若闻道夕死可，
世间何处有高人？

第二天，张真人对外宣布，他从此退隐江湖，不再参与江湖事。瞬时间，整个江湖就像地震了一样，人们没有了主心骨，都不知道该干什么好了。打打杀杀的也不打了，商人也没得吹了。很多商人纷纷上武当来游说张真人，让他出山，包荣华富贵，都被张真人一一回绝了。江湖又恢复了以往的平静，只是这平静似乎预示着一场更加猛烈的暴风雨即将来临。

前仆后继——后来者居上

话说有位少年，自幼好钻研和寻根问底，被人称作“隔一路”。此人不善于口头表达，不会忽悠，但是如果世界只剩下他一个人，那么他便会爆发出神奇的力量。由于内向的性格，他吃了不少亏，但他依然我行我素，并不在乎别人的议论和猜忌甚至是诋毁。这位少年名为无忌。他实际上也确实是无所畏忌，明知山有虎，偏向虎山行，用天真和执著去挑战世俗，跌倒了大不了重来。

既然选择了这条路，就要把它走完。孤独和压迫给了他巨大的动力，每天晚上都在刻苦学习。他学习 IO 大法和磁盘大挪移，学习七星北斗阵和降龙大法。虽然他并没有实践过这些知识，但是依然有一股力量促使他不断的学习钻研。

6.1 初露端倪——外置磁盘柜应用探索

无忌已经充分掌握了前人留下的心法口诀。在不知道该做点什么的时候，他突然有了一个想法。虽然按照七星大侠的 RAID 方式，可以将多块磁盘做成逻辑盘，但是普通的服务器或者 PC 机箱里面，也就安装两三块磁盘，空间就满了。如果做很多块盘的 RAID，把磁盘都放到机箱里面肯定不行，得想个办法来让机器可以带多块磁盘。

“拿出来，拿出来，全部都掏出来！”。他找来一台机器，装了一块 Ultra 320 SCSI 卡，这个卡只有一个通道，可以连接 15 块磁盘。但是 15 块盘怎么放入一个机箱呢？太困难了，

所以必须把这些盘放到机箱外面。但是连线和电源问题又不好办。他索性找来一个箱子，把所有磁盘都放在这个箱子里。箱子有独立电源和散热系统，保障磁盘的稳定运行。接口方面，内部其实就是一条 SCSI 线缆，只不过将它做到了电路板上，然后在外面放一个接口，这个接口是用来连接主机上的 SCSI 卡的。如果主机上装的是不带 RAID 功能的 SCSI 卡，那么加电之后，主机会识别到磁盘箱中的所有磁盘。箱子中有多少磁盘，在 OS 磁盘管理器中就会显示多少块磁盘。如果主机上安装的是带 RAID 功能的 SCSI 卡，那么可以用这个 RAID 卡先来对认到的多块磁盘做一下 RAID，划分出逻辑盘，这时 OS 识别到的就是逻辑磁盘，而不会认到箱子中的物理磁盘。

这种简单的磁盘箱如图 6.1 所示，无忌给它取了个学名，叫做“JBOD”，也就是 Just a Bound Of Disks，“只是一串磁盘”，这个描述非常形象。无忌立即将这个做法公布了出去，没想到大受欢迎，一时间各个厂家争相生产这种磁盘柜，在市场上卖得很火。

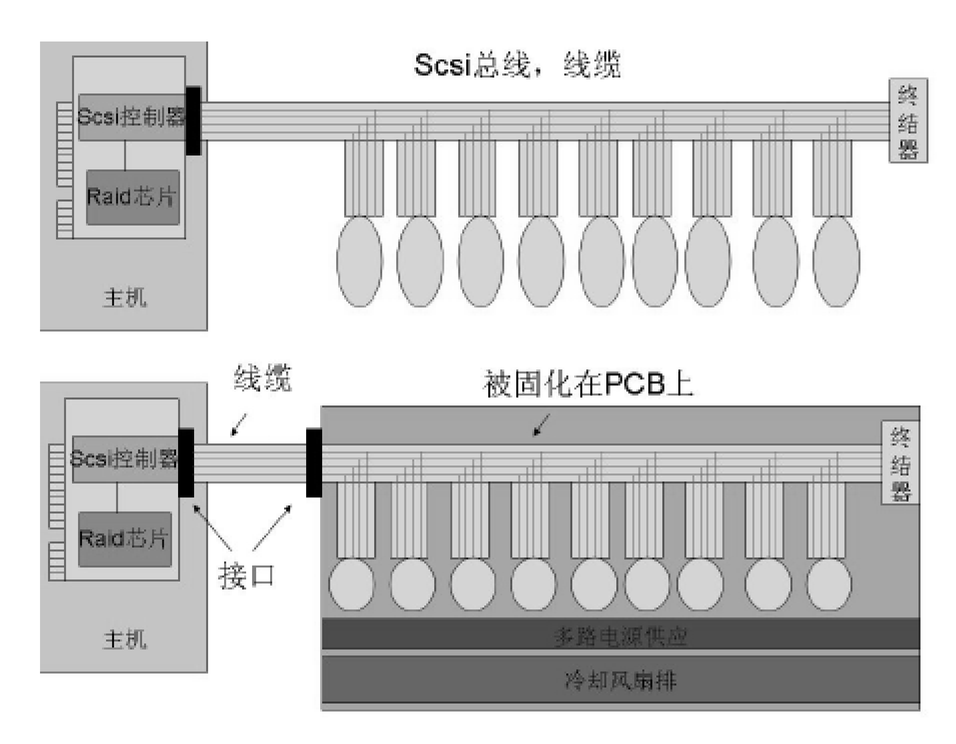


图6.1　JBOD 磁盘阵列示意图

6.2　精益求精——结合 RAID 卡实现外置磁盘阵列

思考

能否把 RAID 功能做到磁盘箱中，因为如果要调整 RAID 的话，还需要重启主机等，会影响主机应用。如果做到了磁盘箱中，那么在主机上就不需要做什么，只要在磁盘箱中做完之后连接到主机，主机重启之后或者不用重启就能认到新逻辑盘了。

经过多次实验，终于做成了一个设备。少年把这种自带 RAID 控制器的磁盘箱叫做“磁盘阵列”。自此在江湖上有了一个不成文的规定，凡是 JBOD 都叫做磁盘柜，凡是自带 RAID 控制器的盘柜就叫做磁盘阵列或者盘阵。盘柜和盘阵，前者只是一串外置的磁盘，而后者自带 RAID 控制器。图 6.2 所示是 JBOD 磁盘柜实物图。

盘阵是在盘柜的基础上，将内部的磁盘经过其自带的 RAID 控制器的分分合合，虚拟化成逻辑磁盘，然后经过外部 SCSI 接口连接到主机上端的 SCSI 接口。此时，整个盘阵对于主机来说，就是主机 SCSI 总线上的一个或者多个设备，具有一个或者多个 SCSI ID。所有逻辑磁盘都以 LUN 的形式呈现给主机。

如图 6.3 所示，盘阵中的 SCSI 控制器在逻辑上有两个部分，右边的 S2 控制器连接了一

条 SCSI 总线，上面有若干磁盘。左边的 S1 控制器同样也连接了一条 SCSI 总线，但是上面只有两个设备，一个就是主机 SCSI 控制器，另一个就是它自己。

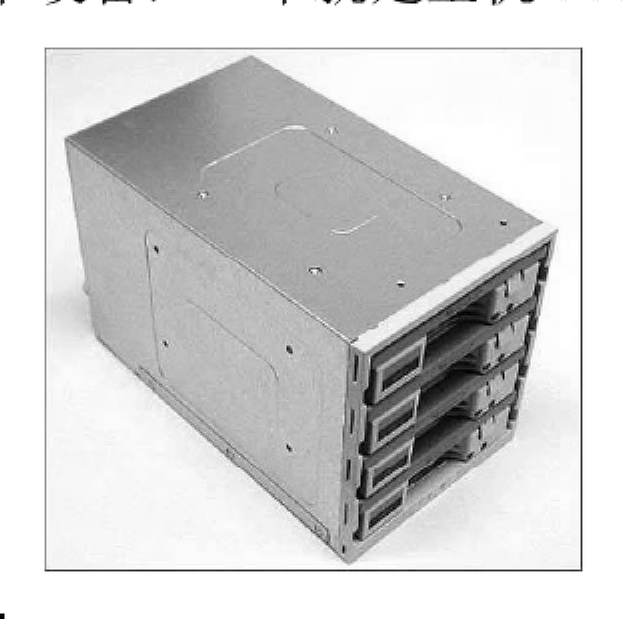


图6.2　JBOD 磁盘柜实物图

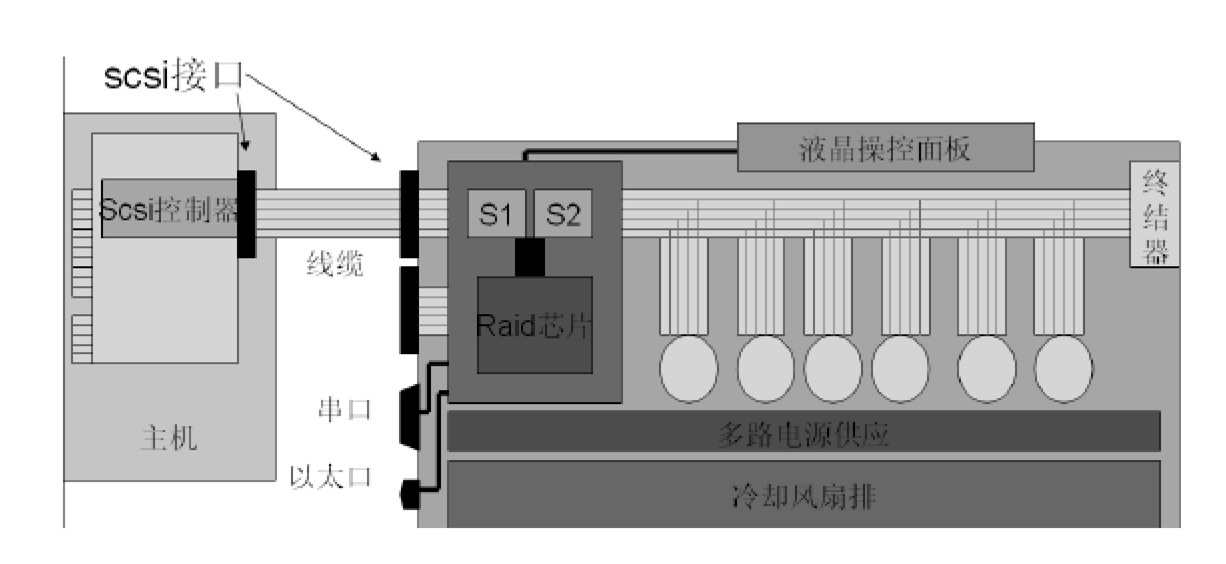


图6.3　带 RAID 控制器的磁盘阵列示意图

毫无疑问，在左边的 SCSI 总线上，盘阵 SCSI 控制器是作为 Target 模式，被主机 SCSI 控制器操控，处于被动地位。在右边的 SCSI 总线上，盘阵的 S2 控制器成了 Initiator 模式，它在右边总线上占据主动权，拥有最高优先级，而各个磁盘均为 SCSI Target，受控于 Initiator。当然 S1 和 S2 不一定就是两块物理上分开的芯片，很有可能就是一块单独的芯片逻辑的分成两个部分。甚至有可能将 RAID 芯片和 SCSI 控制器芯片全部集成到一个大芯片中。

图 6.4 所示的是一个 SATA 盘阵控制器的主板示意图。

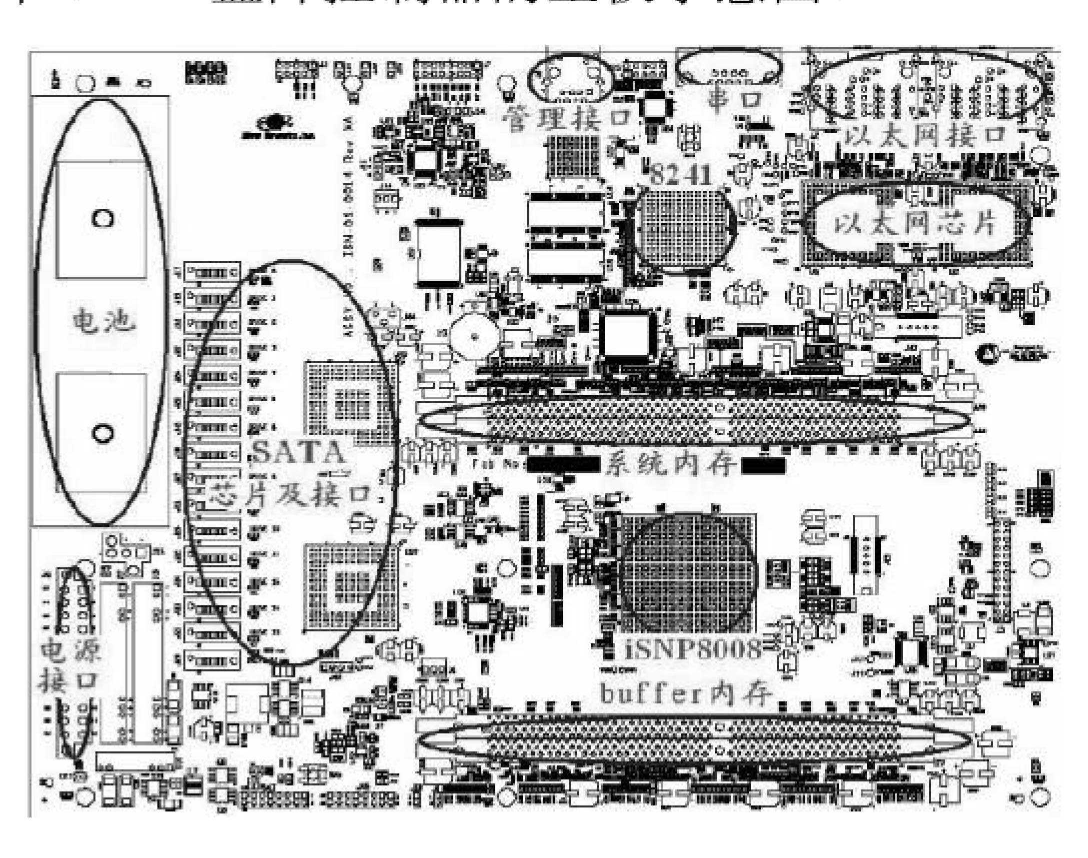


图6.4　一个可以连接 16 块 SATA 磁盘的小型 RAID 控制器主板

图 6.5 所示的是一个小型盘阵控制器的内部实物图。

图 6.6 所示的是一台盘阵的磁盘插槽实物图。

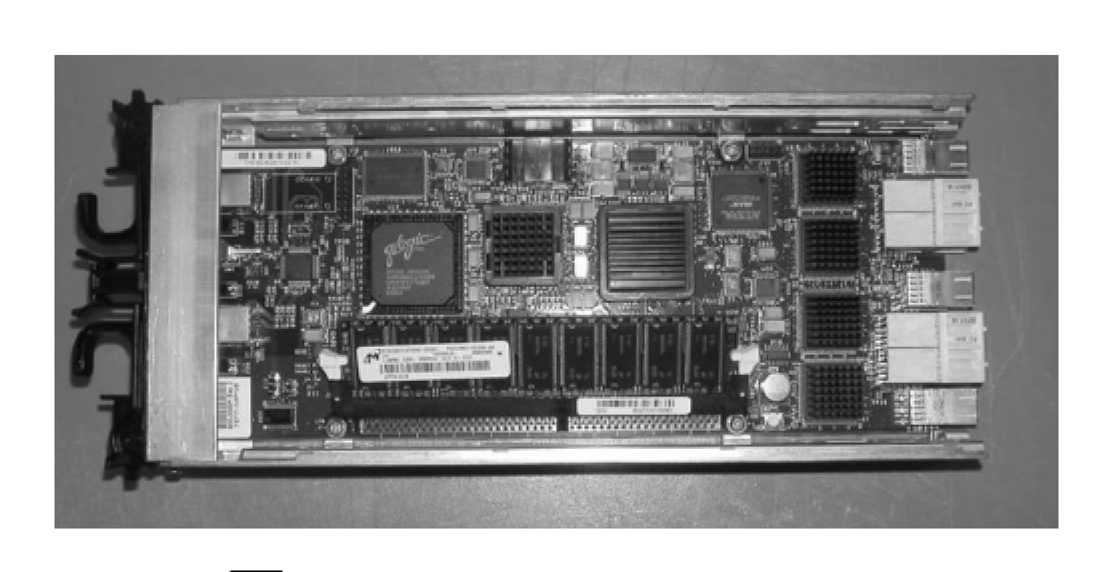


图6.5　一个小型控制器实物图



图6.6　磁盘插槽、背板

图 6.7 所示的为这台盘阵的电源模块插槽。



图6.7 电源模块插槽

6.3 独立宣言——独立的外部磁盘阵列

主机由于肚量太小容不下想法太多的磁盘！终于磁盘从主机内部跑出来了，磁盘们在外置的大箱子里，在 RAID 控制器的带领下，欣欣向荣，勇往直前！

磁盘和控制器发布了独立宣言，彻底摆脱了主机的束缚，成为与主机对峙的一个独立的外部设备。从此以后，存储技术才真正的成为一个独立的庞大学科，并不断发展壮大。本书后面的章节会介绍更多的存储技术，包括存储网络和网络存储。

1. 前端和后端

对于盘阵来说，图 6.3 中 RAID 控制器的左边就称为“前端”，右边则称为“后端”。面向主机对外提供服务的就叫前端，面向自己管理的磁盘用于内部管理而外部不需要了解的部分就叫做后端。同样，对于主机来说，它的 SCSI 适配器反而成了后端，而以太网卡可能变成了前端。因为对于主机来说，直接面对外部客户机的是以太网，而管理磁盘的工作不必对客户说明，所以变成了后端。

2. 内部接口和外部接口

对于盘阵来说，还有一个内部接口和外部接口的概念。内部接口是指盘阵 RAID 控制器连接其内部磁盘时候用的接口，比如可以连接 IDE 磁盘，SCSI 磁盘，SATA 磁盘和 FC 磁盘等。外部接口是指盘阵控制器对于主机端，也就是前端，提供的是什么接口，比如 SCSI 接口、FC 接口等。内部接口可以和外部接口相同，比如内部用 SCSI 磁盘，外部也用 SCSI 接口连接主机，这种情况也就是图 6.3 中所示的情况。

内外接口也可以不同，比如内部连接 IDE 磁盘，外部却用 SCSI 接口连接主机(仅限于盘阵，盘柜必须内外接口一致)。盘阵控制器是一个虚拟化引擎，它的前端和后端可以不一致，它可以向主机报告其有多少 LUN，尽管内部的磁盘是 IDE 的。

3. 多外部接口

同时，也不要被盘阵上为什么可以有多个外部 SCSI 接口而感到困惑。有多个接口是为了连接多台主机用的。每个由盘阵 RAID 控制器生成的逻辑磁盘，可以通过设置只分配(Assign/Map)到其中一个口，比如 LUN1 被分配到了 1 号口，那么连接到 2 号口的主机就

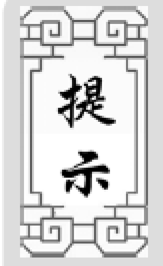
不会看到这个 LUN。也可以把一个 LUN 同时分配(或叫做 Map，映射)到两个口，那么两台主机能同时识别出这个 LUN。让两台主机同时对一个 LUN 写数据，底层是允许的，但是很容易造成数据的不一致，除非使用集群文件系统，或者高可用性系统软件的参与。

4. 关于 LUN

LUN 是 SCSI 协议中的名词，我们前面也描述过。LUN 是 SCSI ID 的更细一级的地址号，每个 SCSI ID(Target ID)下面还可以有更多的 LUN ID(视 ID 字段的长度而定)。对于大型磁盘阵列来说，可以生成几百甚至几千个虚拟磁盘，为每个虚拟磁盘分配一个 SCSI ID 是远远不够用的。因为每个 SCSI 总线最多允许 16 个设备接入(目前 32 位 SCSI 标准最大允许 32 个设备)。要在一条总线上放置多于 16 个物理设备也是不可能的，LUN 就是这样一个次级寻址 ID。磁盘阵列可以在一个 SCSI ID 下虚拟多个 LUN 地址，每个 LUN 地址对应一个虚拟磁盘，这样就可以在一条总线上生成众多虚拟磁盘，以满足需求。

后来，人们把硬件层次生成的虚拟磁盘，统一称为“LUN”，不管是不是在 SCSI 环境下，虽然 LUN 最初只是 SCSI 体系里面的一个概念。而由软件生成的虚拟磁盘，统一称为“卷”，比如各种卷管理软件、软 RAID 软件等所生成的虚拟磁盘。

有些盘阵配有液晶操控面板，如图 6.8 所示。而有些低端的盘阵更是在液晶面板周围加上了按钮，用来对盘阵进行简单快速的配置，比如查看磁盘状态、设置 RAID、化分逻辑磁盘等。这种方式极其简化了配置操作，一般用户通过阅读说明书就可以完成配置。不过液晶屏幕比较小，能完成的功能不多，操作相比用鼠标要麻烦。所以一些盘阵提供了 COM 口或者以太网接口，可以用 PC 机连接这些接口与盘阵通信，通过仿真终端或 Web 界面就可以对盘阵进行配置。



用户用 PC 机与盘阵的 COM 口或专用于配置的以太网接口连接，完全是为了配置磁盘阵列的各种参数，而不是通过这些配置专用接口从磁盘阵列的磁盘上读写数据。

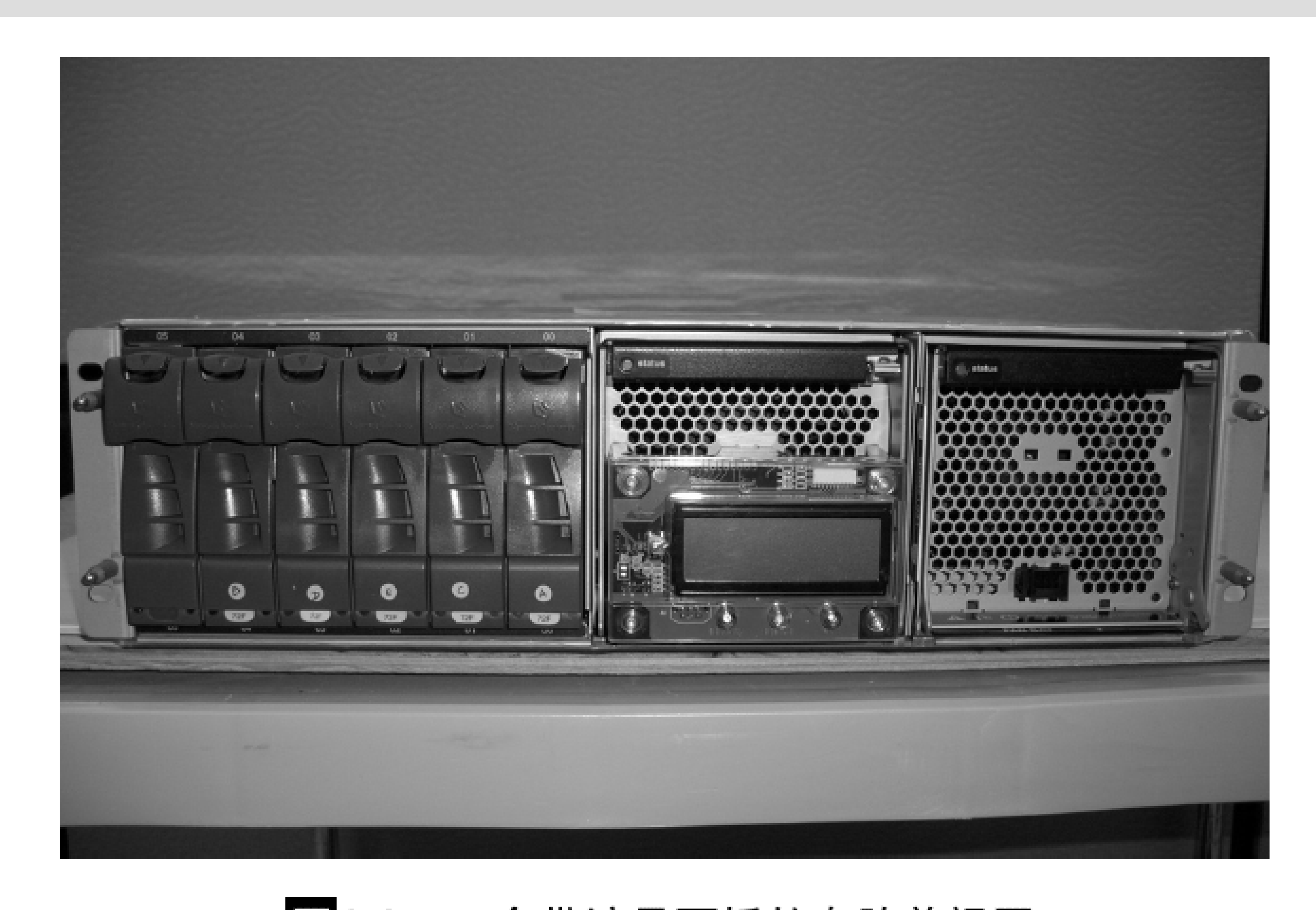


图6.8　一个带液晶面板的盘阵前视图

6.4 双龙戏珠——双控制器的高安全性磁盘阵列

如果盘阵内部只有一个控制器模块，那么会是一个 SPOF(Single Point Of Failure)，即单点故障点。所以一些高端的盘阵内部都有两个控制器，互为冗余。分配给其中一个控制器的 LUN 逻辑卷，可以在这个控制器因故障失效的时候，自动被另一个工作正常的控制器接管，继续处理针对这个 LUN 的读写请求。两个控制器平时都管理各自的 LUN，一旦发现对方故障，那么就会自动将所有 LUN 都接管过来。

因为如此，两个控制器之间需要相互通信，通告对方自己的状态以及交互一些其他的信息。两个控制器之间可以用 PCI 总线连接，也可以用厂商自己设计的总线来连接，没有统一标准。至于交互信息的逻辑和内容，这更是因品牌而不同，更没有标准来统一它们。

为了避免单点故障，给盘阵安装一个额外的控制器，这个控制器和原来的控制器在它们后端共享一条或者多条磁盘总线。两个控制器可以使用 Active-Standby 的方式，也可以使用 Dual-Active 的互备方式。

1. Active-Standby

这种方式又称 HA 方式(High Availability，高可用性)，即两个控制器中同一时刻只有一个在工作，另外一个处于等待、同步和监控状态。一旦主控制器发生故障，则备份控制器立即接管其工作。

对于内部为 SCSI 总线的双控制器盘阵，在机头内部的一条 SCSI 总线中，两个控制器可以分别占用一个 ID，这样剩余 14 个 ID 给磁盘使用。平时只有主控制器这个 ID 作为 Initiator 向除了备份控制器 ID 之外总线上的其他 ID(也就是所有磁盘的 ID)来发送指令从而读写数据。

同时备份控制器与主控制器之间保持通信和缓存同步，一旦主控制器与备份控制器失去联系，那么备份控制器立即接管主控制器。同时为了预防脑分裂(见下文)，备份控制器在接管之前需要通过某种机制将主控制器断电或者重启，释放其总线使用权，然后自己接管后端总线和前端总线。



主机端必须用两个 SCSI 适配器分别连接到盘阵的两个控制器上，才能达到冗余的目的，但是这样做主机端必须通过某种方式感知到这种 HA 策略并在故障发生时切换。目前，由于 SCSI 盘阵本身比较低端，可接入容量不大，所以没有双控制器的设计，以上文字只是对 HA 机制的一种描述。但是对于本书后面要讲述的 FC 盘阵来说，使用双控制器以及在主机端使用双 FC 适配卡是非常普遍的。

2. Dual-Active

顾名思义，这种双控制器的实现方式是两个控制器同时在工作，每个控制器都对所有后端的总线有通路，但是每个总线平时只被其中一个控制器管理，另一个控制器不去触动。可以将后端一半数量的总线交由一个控制器管理，另一半交由另外一个控制器管理。一旦

其中一个控制损坏，则另外一个控制器接管所有总线。这种方式比 Active-Standby 方式高效很多。

3. 脑分裂(Split Brain)

这个词明显有点恐怖。设想一下，如果某时刻连接两个控制器之间的通路出现了问题，而不是其中某个控制器死机，此时两个控制器其实都是工作正常的，但是两者都检测不到对方的存在，所以两者都尝试接管所有总线，这时候就是所谓的“脑分裂”，即同时有两个活动控制器来操控所有后端设备。这种情况是可怕的，类似精神分裂症。

如何预防这种情况呢？通常做法是利用一个仲裁者来选择到底使用哪一个控制器接管所有总线，比如用一两个控制器都能访问到的磁盘，控制器向其上写入自己的仲裁信息。一旦发生脑分裂，二者就参考这个磁盘，谁最后写入了信息就把控制权给谁。或者用一种电源控制器，一旦其中某个控制器要接管，那么不管对方是确实发生故障了还是正常的，这个控制器都会向电源控制器发送信号，让对方重启并进入 Standby 状态，这样就成功的预防了脑分裂。

接管了总线的控制器一般都会对总线上所有磁盘进行 SCSI Reserve 操作，即预订操作。总线上所有目标设备一旦被预订，它们便不再接受其他控制器的 IO 请求。SCSI 2 标准中的 SCSI Reserve 不允许其他控制器读写被原有控制器预订的设备，但是 SCSI 3 中的 Reserve 策略有了一些灵活性，可以允许其他控制器对已经被预订的目标设备进行读 IO，而写 IO 则被拒绝。

图 6.9 所示的是一双控制器盘阵机头示意图。

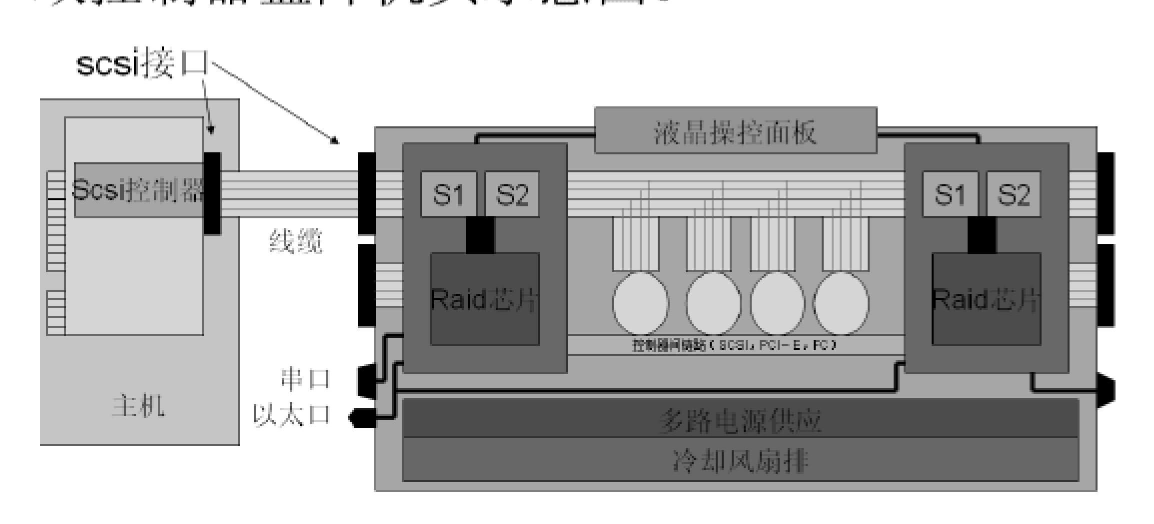


图6.9　双控制器磁盘阵列示意图



实际中，由于 SCSI 盘阵比较低端，一般没有这种设计模式的产品。

6.5　龙头凤尾——连接多个扩展柜

一条 SCSI 总线最多就可以连接 15 块磁盘，为了这 15 块磁盘，大动干戈的赋予两个昂贵的 RAID 控制器，有点不值。为了把这两个控制器充分的利用起来，榨取最后一滴性能，15 块磁盘不够，那就再加。我们前面说过，一个控制器上可以有多个通道，一个通道下面就是一条 SCSI 总线，那么将盘阵的每个控制器上再多接一个或者两个通道，来充分发挥它的能力，这样就比较实惠了。如图 6.10 所示，这台盘阵机头带有一个扩展后端磁盘柜接口。

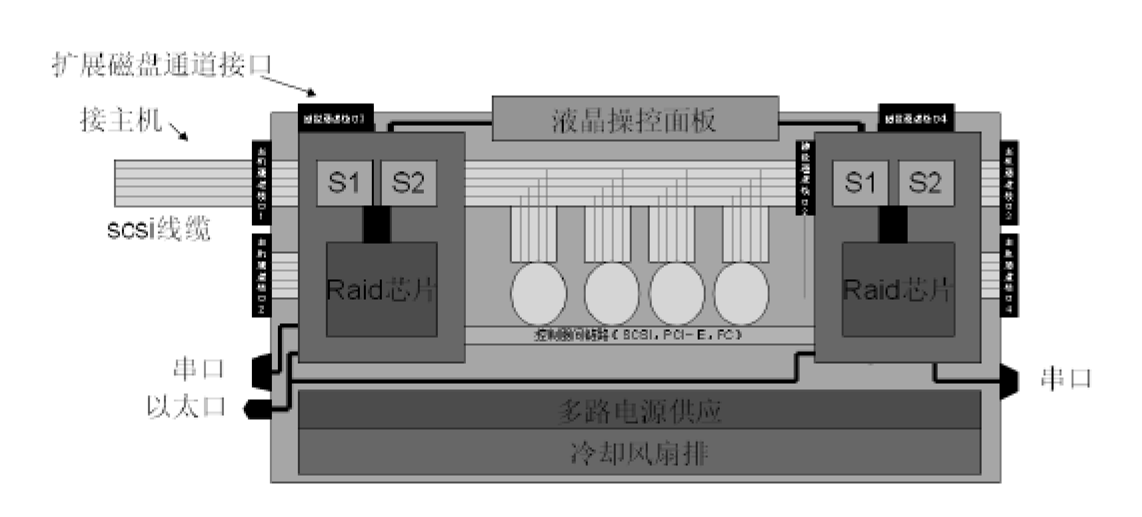


图6.10　带有一个扩展外部磁盘通道接口的控制器示意图

通道建好之后，当然下一步就是要扩充磁盘数量了。当然，JBOD 就成了最佳选择。

图 6.11 所示的盘阵的每个控制器上多出一个额外的磁盘通道接口，这个接口露在机箱外面，用线缆连接了一个 JBOD 扩展柜。

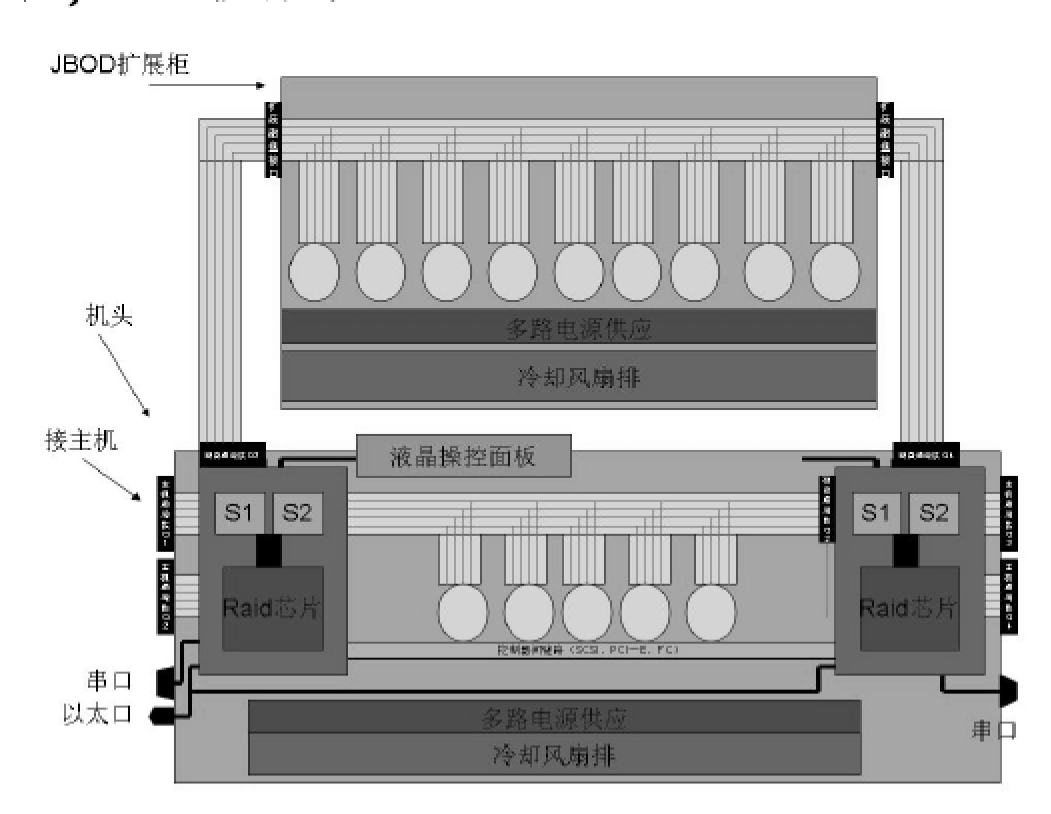


图6.11　外接一个 JBOD 扩展柜的磁盘阵列

经过这样的改造，可连接的磁盘数量成倍增长。图中所示的是每个控制器增加了一个磁盘通道，还可以增加到两个或者多个通道。理论上，只要 RAID 控制器处理速度够强，总线带宽和面板上空间够大，多增加几个通道都没问题。

JBOD 盘柜以前只有一个外部接口，为了配合双控制器，JBOD 在其外部也增加了一个接口用来连接冗余的控制器。这样，扩展柜上也有两个外部接口了。

把带有控制器的磁盘柜称作“机头”，因为它就像火车头一样，是提供动力的。机头里可以有磁盘，也可以根本不含磁盘。把用于扩展容量用的 JBOD 叫做“扩展柜”，它就像一节节火车车厢，本身没有动力，全靠车头带，但是基本的供电和冷却系统还是要有的。图 6.12 所示的是 IBM 的 DS400 盘阵机头后视图，它是每控制器提供 3 个通道，机头内部的磁盘占用一个，然后另外 2 个提供扩展，在后面板上给出两个 SCSI 接口。图中“Expand ports”所示的就是这两个 SCSI 接口。右边空白的地方是用来接入另外一个控制器的，这个控制器是可选组件。

图 6.13 是用于连接 DS400 机头的扩展柜 EXP400。可以看到它的左右各有一个接口模块，每个模块上有一个 SCSI 接口用来连接机头。

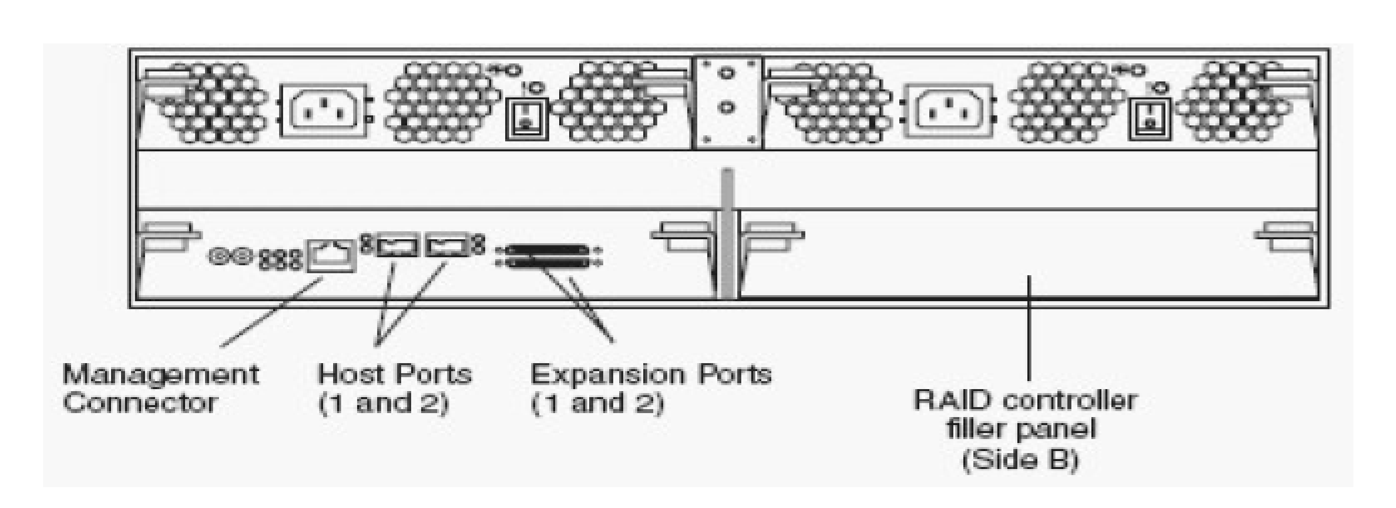


图6.12 DS400 盘阵的机头后视图

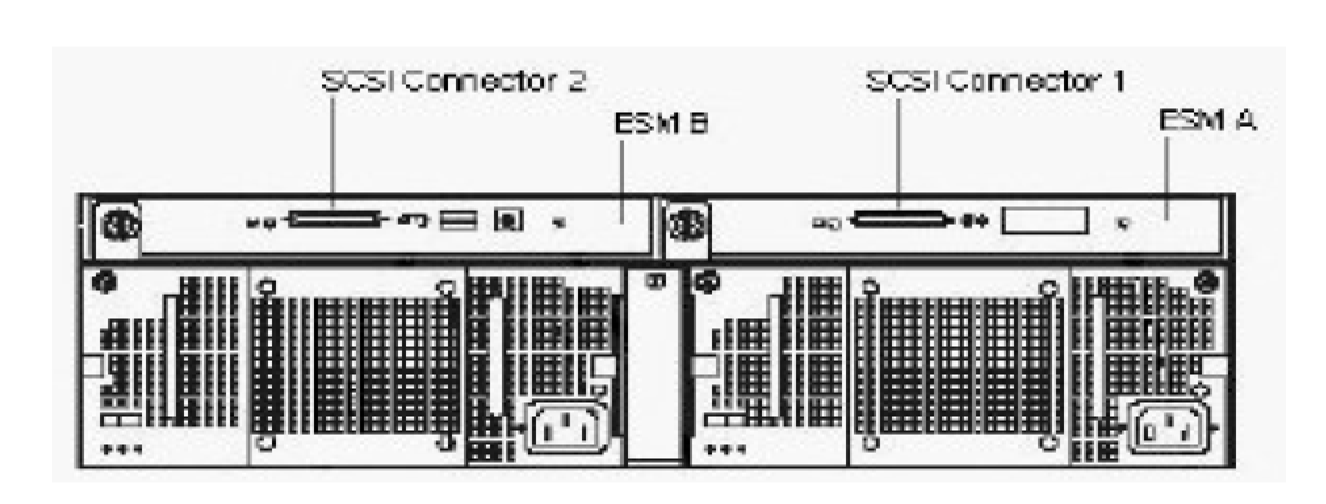


图6.13 EXP400 扩展柜的后视图

6.6 锦上添花——完整功能的模块化磁盘阵列

再后来，机头做的都比较漂亮，而是感觉很厚实。但是 JBOD 就是一堆磁盘，显得和机头有些不搭配。所以也给扩展柜增加了所谓的模块，外观不仔细看的话，和机头没很大的区别。只不过扩展柜的模块上，没有 RAID 控制器的功能，但是会加上一些其他功能，如探测磁盘温度等二线辅助功能。这个模块将接口、功能芯片、电路等都集成在一个板子上，所以外观和机头差不多。

图 6.9 所示的 ESM 模块，就是实现这些功能的插板。图 6.11 中所示的是一个磁盘扩展柜的实物后视图，可以看到上下两个模块，这两个模块不但负责链路通信，还负责收集设备各处的传感器发来的信息。

图 6.14 为一台盘阵的前视图。图 6.15 是一台 FC 接口的扩展柜后视图，可以看到上下两个 ESH(Electrical Switch Hub)模块。这些磁盘扩展柜上的模块中主要包含单片机或者 DSP 芯片、FC-AL 半交换逻辑处理以及其他功能的 FPGA/ASIC/CPLD 芯片、SFP 适配器编码芯片、ROM 或者 Flash 芯片(存放 Firmware)、RAM 缓存芯片(用于存放芯片执行程序时所需的数据)等，视设计不同而定。如果有新的 Firmware 被开发出来，可以将程序逻辑写入 Flash 或 ROM 芯片中，这个过程就是固件升级。FPGA/CPLD 等芯片需要用外置的编程器写入新的电路逻辑。ASIC 芯片不可升级，是固定逻辑的芯片，适用于成熟的、量产的芯片，比如 SFP 编码芯片等。



图6.14 一个扩展柜的前视图

图 6.16 所示为扩展柜上的一个 ESH2 模块的内部实物图。

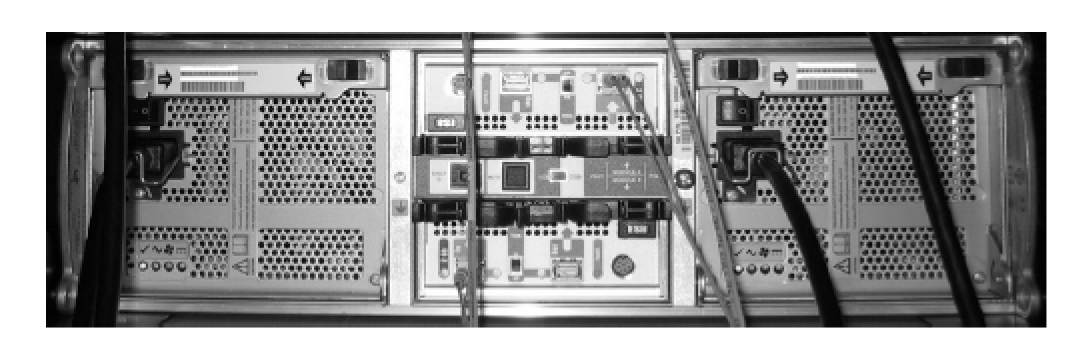


图6.15 一个磁盘扩展柜的后视图



图6.16 ESH2 模块实物图

6.7 一脉相承——主机和磁盘阵列本是一家

1. 盘阵控制器的主机化

随着人们需求不断提高，一个存储系统拥有几 TB 甚至几十几百 TB 的容量已经不是什么惊人的事情了。面对如此大的容量和如此多的磁盘，小小的控制器已经不能满足要求了。因此大的主机系统替代了短小精悍的控制器。



可能有人已经不知所措了，用主机系统替代盘阵控制器，这不矛盾么？盘阵是给主机服务的，主机替代了盘阵控制器，岂不是乱了辈分了？

事实并非如此。我们知道，主机系统的经典架构就是 CPU、内存、总线、各种 IO 设备和 CPU 执行的代码(软件)。而观察一下盘阵控制器的基本架构：RAID 控制器芯片(CPU)、内存、总线、IO 接口(SCSI 接口等)和 RAID 芯片执行的代码(软件)。可以说盘阵控制器就是一个简单的主机系统。

既然这样，完全可以用一台主机服务器来充当存储系统的控制器。比如，在这台主机上插入几张 SCSI 卡作为前端接口卡，然后再插入若干 SCSI 卡作为后端连接磁盘箱的接口卡，然后设计软件从/向后端读写数据，经过处理或者虚拟化之后，再传送给前端的主机服务器。

目前有两种趋势，一种是趋向使用现成的主机来充当控制器的载体，另一种是趋向使用高集成度的芯片作为控制器的核心。两种趋势各有利弊。

图 6.17 所示的是一台主机化的磁盘阵列实物图。

2. 盘阵的类型

按照前端和后端接口来分，有 SCSI-FC 盘阵，FC-FC 盘阵，SATA-FC 盘阵，SCSI-SCSI 盘阵等类型。SCSI-FC 类型表示后端接口为 SCSI 接口，前端用于连接主机的为 FC 接口，也就是后端为 SCSI 磁盘，前端为 FC 接口的盘阵。

我们在后面的章节会讲到 FC-FC 盘阵，这也是目前最高端的盘阵所采用的架构。图 6.18 所示的就是一台大型 FC 磁盘阵列的透视图图示中一共 5 个机柜，中间的机柜整柜都为控制器，上方可见一排 IO 插卡，插卡上方为 9 个风扇。其余机柜中均为磁盘扩展柜。



图6.17 主机化的盘阵控制器

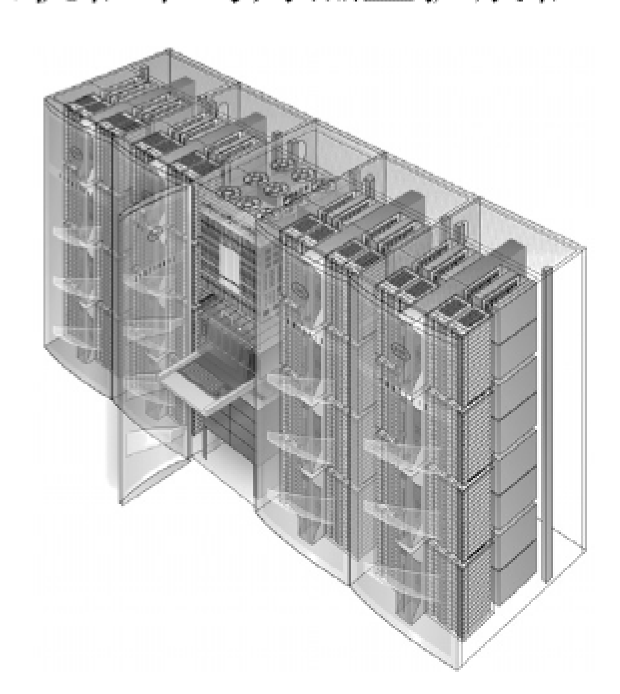


图6.18 EMC DMX 系列盘阵透视图

6.8 天罗地网——SAN(Storage Area Network)存储区域网络

大家来看最后一张图片，如图 6.19 所示。我们一开始描绘的那张“网中有网”的图片，现在大家应该能更深刻的理解了。网络，不仅仅指以太网，TCP/IP 网，他还可以是 SCSI 网，PCI 总线网，USB 网等。RAID 控制器，就相当于一个路由器，也就是协议转换器。

将磁盘放到了主机外部，存储设备和主机之间，就形成了又一个独立的网络：存储区域网络(Storage Area Network，SAN)。

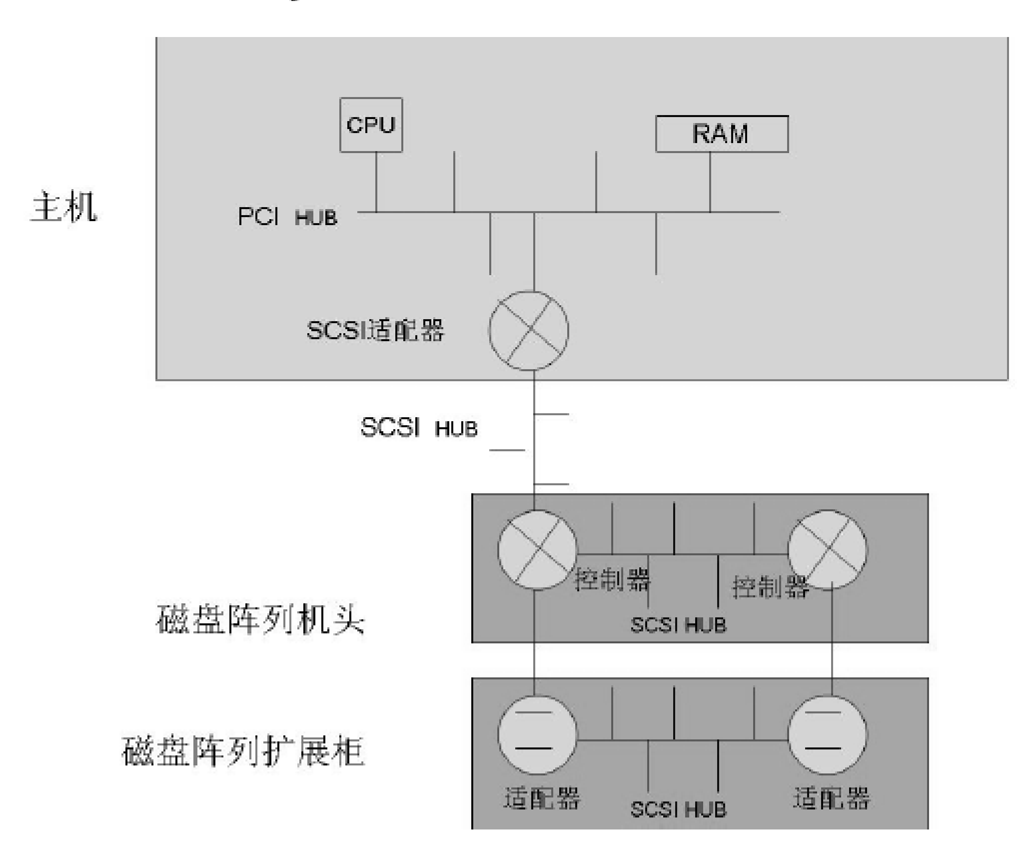
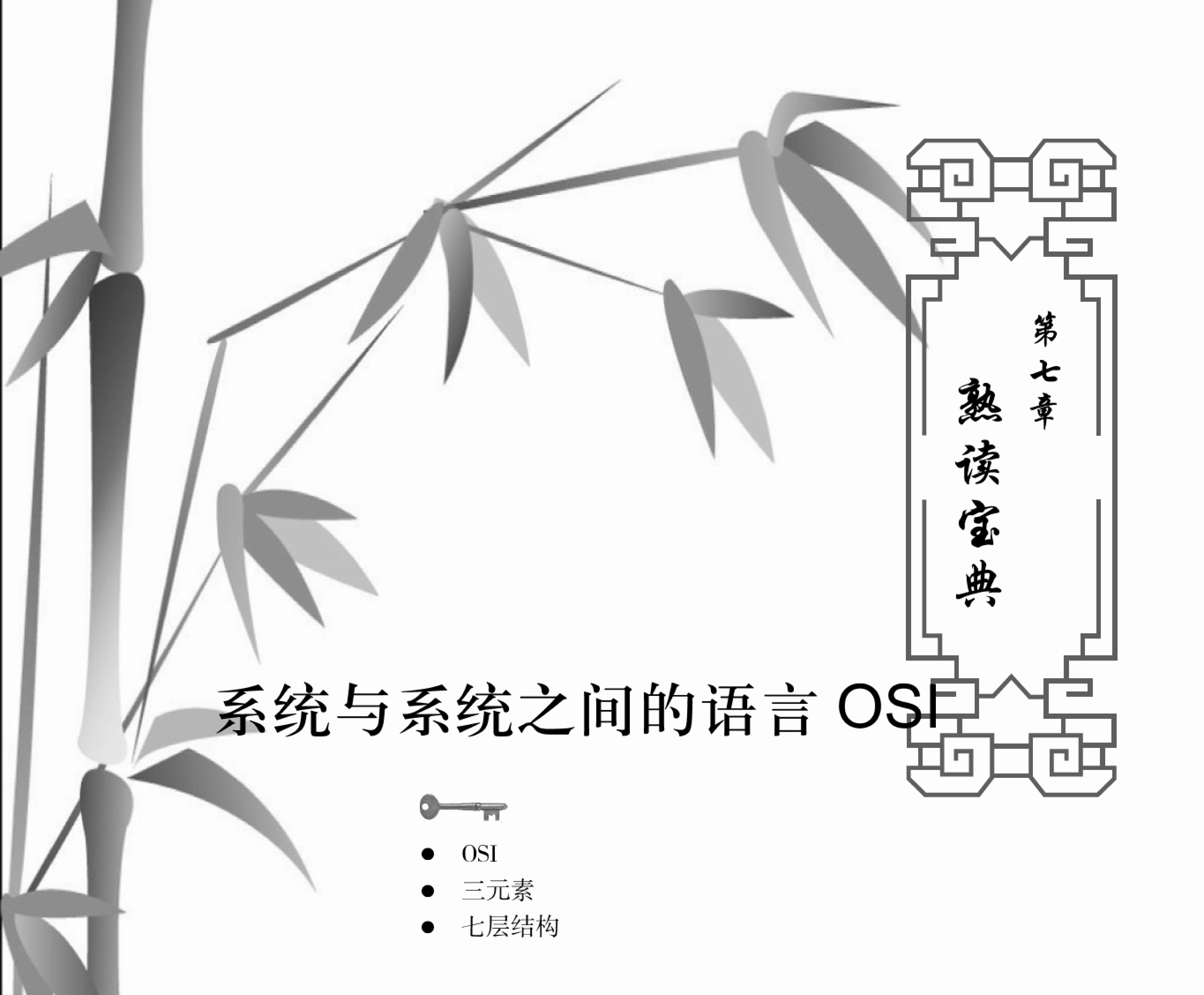


图6.19 网中有网

数据就是在这种网络中来回穿梭，格式不断被转换和还原。



第七章 熟读宝典

系统与系统之间的语言 OSI

- OSI
- 三元素
- 七层结构

千百年来，江湖中人一直都把祖宗流传下来的一本宝典，铭记于心！这本宝典就是号称“号令武林，莫敢不从”的 OSI 大典！

任何系统之间，如果需要通信，就有一套自己的协议系统。这个协议系统不仅要定义双方互相通信所使用的语言，还要规定所使用的硬件，比如通信线路等。例如以太网协议，凡是接入以太网的 (交换机或者 HUB)节点，都必须遵循以太网所规定的通信规程。两个对讲机之间进行通话，必须预先定义好发送和接收的频率，而且还要指定通信的逻辑，比如每说一句话之后，要说一个“完毕”，表示本地已经说完，该对方说了。

7.1 人类模型与计算机模型的对比剖析

人和计算机，二者有着天然的相似性。

7.1.1 人类模型

人类自身用语言来交流信息，这本身就是系统间通信的极佳例子。每个人都是一个系统，这个系统由八大子系统组成，每个子系统行使自己的功能，各个子系统相互配合，使得人体可以做出各种各样的事情，包括制造计算机系统这个“部分仿生”产物。

1. 消化系统

负责食物的摄取和消化，使我们获得糖类脂肪蛋白质维生素等营养，再经过一系列生化酶促反应，最终可以生成能量，以 ATP 分子的形式向各个细胞供应化学能，再经过一系列的分子机器(蛋白质或者蛋白质复合体)的处理，可以形成机械能、热、光、电等各种能量形式。

2. 神经系统

负责处理外部信息和调控人类自身运动，使我们能对外界的刺激有很好地反应，包括学习等重要的活动也是在神经系统控制下完成的。比如皮肤、耳朵、眼睛等接收的各种信号，都会经过神经网络传送到大脑进行处理并做出反映。

3. 呼吸系统

气体交换的场所，使人体获得新鲜的氧气。人类制造、储存和利用能量的每个过程，都需要氧气的参与，氧气是很好的氧化剂，人类利用这种氧化剂，氧化摄取的糖、蛋白质等物质，产生能量，如果没有氧气，则人体一切的生命活动就会终止，包括心肌的收缩和扩张运动。

4. 循环系统

负责氧气、营养和体液以及各种信号控制分子的运输、废物和二氧化碳的排泄，以及免疫活动。人体的这些空间中，各种器官按照规则分布着，每个器官行使其功能，必须供给它们能量、水和氧气以及各种控制因子蛋白，遍布周身的密密麻麻的动脉、静脉和毛细血管，就是运输这些物质必须的通道。这些物质溶在血液中，从动脉流入器官，完成生命逻辑后，从静脉流入肾脏，过滤废物，排泄，干净的静脉血再流回心脏，然后进入动脉，经过肺部的时候，将肺部获取的氧气吸溶入血液，成为鲜红色的动脉血，然后再流入各个器官，供给必须物质。

5. 运动系统

负责身体的活动，使我们可以做出各种姿势。骨骼和肌肉属于运动系统，骨骼虽然不直接运动，但是它能在肌肉的牵引下进行运动。控制肌肉运动的是神经信号，大脑发出神经控制信号(一系列的蛋白质或者离子流)，肌肉组织收到之后，便会引发雪崩似的化学反应，

包括肌肉细胞中的微管结构的不断重组和释放，用这种形式来改变细胞形状，从而造成肌肉收缩或者扩张。而微管的组装和释放，需要能量和酶的参与，这时候，细胞便会贪婪的利用 ATP 分子来获取化学能，而血液 ATP 损耗之后，血液便会加速流动到肌肉周围，提供更多的能量。这也是运动时心脏加速跳动的原因。而心脏跳动速度的控制因素很多，比如受到惊吓的时候，大脑便会分泌一些物质，传送到心肌，引发反应，跳动速度加快。

6. 内分泌系统

调解生理活动，使各个器官组织协调运作。内分泌系统是一个中央调控系统，比如生长素就是内分泌系统分泌的一种蛋白质分子，它被骨骼吸收后，可以促进骨骼生长。内分泌系统如果发生功能故障，则人体就会表现异常。包括精神和物理上的异常。

7. 生殖系统

负责生殖活动，维持第二性征。生殖系统是人类繁衍的关键，生殖系统将人类的所有特征融入精子和卵子，受精卵在母亲子宫中逐渐发育成全功能的人体。

8. 泌尿系统

负责血液中生化废物的排泄，产生尿液。

7.1.2 计算机模型

我们发现，计算机系统和工作模式，与生物系统有很大一部分是类似的。

1. 计算机的消化系统

消化子系统是为整个系统提供基本能量和排泄消化废物的。给计算机提供能量的是电源和密布在电路板上的供电线路。为计算机排泄能量废物的，是接地低电位触点。电流流经供电线路，到接地触点终止，将能量提供给电路板上的各个部件(器官)。此外，计算机的“消化”也包含另外的意思，即吞入数据，吐出数据的过程。计算机从磁盘上读入数据，进行计算(消化)后，再输出数据。

2. 计算机的循环系统

循环子系统是为整个系统提供能量和物质传输通道的。给计算机提供数据传输通道的是各种总线，数据从总线流入 CPU 处理单元，完成计算逻辑后，数据又从总线输出到内存或者外设。所以我们说，计算机的循环系统，就是总线，心脏就是电路振荡装置。

3. 计算机的呼吸系统

呼吸子系统是为整个系统提供完成生命逻辑所必需的氧化剂，即氧气为计算机散热的风扇，貌似呼吸系统，但风扇的功能更类似于皮肤和毛孔的散热作用。

4. 计算机的神经系统

神经子系统的作用是传输各种信号，调节各个器官的功能。对于生物体来说，神经系统是运行在脑组织中的一种逻辑，这种逻辑通过执行一系列生化物理反应来体现它的存在。通过向血液中释放各种蛋白质信号分子，从而靶向各种器官，调节它们的功能。对于计算

机系统来说，神经系统就是由 CPU 载入执行的程序，程序生成各种信号，通过总线传输给各个外部设备，从而调节它们的工作。

计算机的神经系统，可以认为就是外部硬件设备的驱动程序。神经网络就是控制总线，循环系统是数据总线。而生物体内没有地址总线的概念，血管凌乱的分布，并没有一种显式的区分机制来区分各个器官或者组，信号分子可以遍布周游全身。

5. 计算机的运动系统

计算机本身是一堆电路和芯片，不存在运动的概念。但是如果向计算机接口上接入了可以运动的部件，比如打印机、电动机、硬盘等，那么这些设备就可以在计算机控制信号(神经信号)的驱动下做运动，并且可以打印或者读写数据。

6. 计算机的生殖系统

目前，计算机系统表现的生殖能力，只是在一个硬件中生成不同的软件。软件通过 CPU 的执行，可以任意复制自身，并可以形成新的逻辑。在这种逻辑下，程序通过无数次的复制，难免在一些细微的 Bug 或者电路干扰的情况下发生奇特的变化，这些变化一开始可能不太会表现出来，但是随着量变的积累，就会引发质变，发生进化。

当然，计算机系统完全可以物理复制硬件，即通过程序控制外部机器，来生产新的计算机硬件，然后将软件复制到硬件上，继续繁殖。

7.1.3 个体间交流是群体进化的动力

人也好，计算机也好，他们之间都在不停地交流着。人和人的交流，让人类得到了进化。同样，计算机之间的交流，也会让计算机得到进化。交流是进化的动力，不可能有某种事物会完全脱离外界的刺激而自身进化。

OSI 便是这种交流所遵循的一张蓝图。

7.2 系统与系统之间的语言——OSI 初步

OSI 模型是一种被提取抽象出来的系统间通信模型。OSI 中文的意思为“开放式系统互联”模型，是一个描述两个或者多个系统之间如何交流的通用模型。它不只适合于计算机系统互联，而且适合任何独立系统之间的互联。比如，人体和人体之间的通信，或者人体和计算机之间的通信，都可以用 OSI 模型来描述。

比如我和你之间需要交流，我们面对面坐着，此时我有一句话要和你说：“您好，您怎么称呼？”。

首先，我要说出这句话，要在脑海中生成这句话，即在语言处理单元中根据要表达的意思，生成符合语法的数据。然后通过神经将数据信号发送到声带、咬肌，舌头，舌头和口形固定之后，使声带振动。声带振动导致口腔空气共振，发出声音，经过空气机械波振动，到达你的耳膜接收器，耳膜被机械波谐振于一定频率，耳膜的振动通过神经信号传导到大脑，大脑相关的处理单元进行信号的解调，最终从神经信号中提取出我说的话“您好，您怎么称呼？”，这句话在大脑中，可能是离子流产生的模拟电信号，也可能是通过其他

形式表示，比如一套编码系统。这句话被传送到大脑的语言处理单元，这个单元分析这句话，“您好”是一个问候语，当你知道我在问候你的时候，你的大脑会在这个“您好”信号的刺激下，进入一种“礼貌”逻辑运算过程，运算生成的信号，通过神经传送到你的颈部肌肉，收缩，使你的头部下降，并使面部肌肉收缩，这就完成了点头致意和微笑的动作。

这个过程，与计算机网络通信的模型完全一致。两台计算机之间通过以太网交换机相连，它们之间要进行通信。比如，a 计算机想向 b 计算机发送一个数据包，这个数据包的内容是“打开文件 C:\tellme.txt”，过程如下所示。

1】 a 计算机首先要在内存中通过双方定义的语言，生成这个数据包，

2】 将这个数据包通过总线发送给 TCP/IP 协议处理单元，告诉 TCP/IP 处理单元对方的 IP 地址和所用的传输方式(UDP 或 TCP)和端口号。

3】 TCP/IP 处理模块收到这个包之后，将它包装好，通过总线发送给以太网卡。

4】 以太网卡对数据包进行编码，然后通过电路将包装好的数据包变成一串电路的高低电平振荡，发送给交换机。

5】 交换机将数据包交换到 b 计算机的接口。

6】 b 计算机收到这串电位流后，将其输送到以太网卡的解码芯片，去掉以太网头，之后产生中断信号，将数据包送到内存。

7】 由 TCP/IP 协议处理模块对这个数据包进行分析，提取 IP 头和 TCP 或 UDP 头，以便区分应输送到哪个应用程序的缓冲区内存。

8】 最终 TCP/IP 协议将“打开文件 C:\tellme.txt”这句话，成功输送到了 b 计算机应用程序的缓冲区内存中。

9】 b 计算机应用程序提取这句话，分析它的语法，发现 a 计算机要求它打开 C:\tellme.txt 文件，则应用程序根据这个命令，调用操作系统打开文件的 API 执行这个操作。

分析一下上面的过程，我们发现如下内容。

1】 数据总是由原始直接可读状态被转变成电路的电位振荡流，或者频率和振幅不断变化的机械波，也可能转换成一定频率的电磁波。

2】 互相通信的两个系统之间必定要有连通的介质，空气、以太网或者其他形式，电磁波传递不需要介质。

3】 相互通信的双方必须知道自己是在和谁通信。

以上三个要素，就是系统互联通信所具备的“连、找、发”三要素。

- 连：就是指通信的双方必须用某种形式连通起来，否则两个没有任何形式连通的系统之间是无法通信的。即便是电磁波通信，也至少通过了电磁波连通。
- 找：是说通信的双方或者多方，必须能够区分自己和对方以及多方(广播系统除外)。
- 发：定义了通信的双方如何将数据通过连通介质或者电磁波发送到对方。

7.3 OSI 模型的七个层次

网络通信三元素抽象模型是对 OSI 模型的更高层次的抽象。OSI 模型将系统间通信划分成了七个层次。

OSI 模型的最上面的三层，可以归属到应用层之中，因为这三层都不关心如何将数据传送到对方，只关心如何组织和表达要传送的数据。

7.3.1 应用层

应用层是 OSI 模型的最上层，它表示一个系统要对另一个系统所传达的最终信息。比如“您好，您怎么称呼？”这句话，就是应用层的数据。应用层只关心应用层自身的逻辑，比如这句话应该用什么语法，应该加逗号还是句号？末尾是否要加一个问号？用“你”还是“您”等这样的逻辑。对于计算机系统来说，上文所述的例子中“open file C:\tellme.txt”，这条指令，就是应用层的数据。应用层程序不必关心这条指令是如何传达到对方的。

7.3.2 表示层

表示层就是对应用层数据的一种表示。如果前面说的“您好，您怎么称呼？”这句话是有一定附加属性的，例如“您好”这两个字要显示在对方的屏幕上，用红色显示在第 1 行的中央，而“您怎么称呼？”这几个字用蓝色显示在第 10 行的中央。这些关于颜色、位置等等类似的信息，就构成了表示层的内容。

发送方必须用一种双方规定好的格式来表示这些信息，比如用一个特定长度和位置的字段来编码各种颜色(一般用三原色的组合编码来表示)，用一个字段来表示行列坐标位置。将这些附加表示层信息字段放置于要表达的内容的前面或后面，接受方按照规定的位置和编码来解析这些表示层信息，然后将颜色和位置信息赋予“您好，您怎么称呼？”这句话，显示于屏幕上。需要强调一点，表示层不一定非得是单独的一个结构体，它可以嵌入在实体数据中。这就是表示层，一些加密等操作就是在表示层来起作用的。

7.3.3 会话层

顾名思义，会话层的逻辑一定是建立某种会话交互机制。这种交互机制实际上是双方的应用程序之间在交互。它们通过交互一些信息，以便确定对方的应用程序处于良好的状态中。比如两个人通电话，拨通之后这个问：“能听清么？”，那个说：“能听清，请讲”，这就是一个会话的过程。也就是说通信的双方在发送实际数据之前，先建立一个会话，互相打个招呼，以便确认双方的应用程序都处于正常状态。

应用层、表示层和会话层的数据内容被封装起来，然后交给了我们的货物押运员传输层。

TCP/IP 协议体系模型中有 4 层，即应用层(应用访问层)、传输层、网络层和物理链路层(硬件访问层)。TCP/IP 协议体系没有完全按照 OSI 匹配，它将 OSI 中的应用层、表示层和会话层统统合并为一层，叫做应用访问层，意思是指这个层全部是与应用程序相关的逻辑，与网络通信无关，应用程序只需调用下层的接口即可完成通信。

7.3.4 传输层

可以说 OSI 模型中上三层属于应用相关的，可以划入应用层范围，而下四层就属于网

络通信方面的。也就是说，下四层的作用是把上三层生成的数据成功的送到目的地。典型的传输层程序如下。

1】 TCP 协议的作用就是保障上层的数据能传输到目的地。TCP 就像一个货运公司的押运员，客户给你的货物，就要保证给客户送到目的地，而不管你通过什么渠道，是直达(直连路由)还是绕道(下一跳路由)，是飞机还是火车、轮船(物理线路类型)。

2】 如果运输过程中出现错误，必须重新把货物发送出去。每件货物到了目的地，必须找收件人签字(TCP 中的 ACK 应答包)，或者一批货物到达后，收件人一次签收(滑动窗口)。

3】 最后回公司登记。



TCP 还处理拥塞和流量控制。比如调度(路由器)选择了走这条路，但是太拥挤了，那么我也不好说什么，因为选哪条路到达目的是由调度(路由器)说了算，我只管押运。那么我只能通知后续的货物慢一点发货，因为这条路太挤了。当道路变得畅通时，我会通知后面的货物加速发货。这就是 TCP 的任务。TCP 是通过接收方返回的 ACK 应答数据包来判断链路是否拥挤，比如发了一批货，半天都没接收到对方的签字，证明链路拥塞，有货物被丢弃了，那么就减缓发送速度。当有 ACK 被接收到后，我会增加一次发送货物的数量，直到再次拥塞。那么调度怎么知道这些货物是送到哪里的呢？这是网络层程序的任务。

注意

传输层的程序一定要运行在通信双方的终端设备上，而不是运行在中间的网络互联设备上。传输层是一种端到端的保障机制，所谓端到端的保障就是指数据从一端发送到另一端之后，对方必须在它的传输保障时间中成功收到并处理了数据，才能算发送成功。如果只是发送到了对方的网卡缓冲区，此时发生故障，如突然断电，这就不叫端到端的保障。因为数据在网卡缓冲区内，还没有被提交到 TCP 协议的处理逻辑中进行处理，所以不会返回成功信号给发送方，那么这个数据包就没有被发送成功，发送方会通过超时来感知到这个结果。

7.3.5 网络层

客户把货物交给货运公司的时候，必须填写目的地址(比如 IP 地址)。只要一个地址就够了，至于到这个地址应该做几路公交车或哪趟火车等问题，客户统统不管，全部交给网络层处理。

1】 货运公司为每件货物贴上一个地址标签(IP 头)。

2】 货运公司的调度们掌握了全球范围的地址信息(路由表)，比如去某某地方应该走哪条路。

3】 在选择了一条路之后，就让司机开车上路了。

4】 押运员进行理货和收发货物，没事就在后车厢里睡觉。

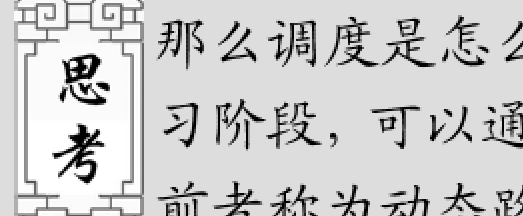
5】 此时最忙的是各个中转站的调度了。货物每次中转到一个地方就交给那个地方的调度，由那个调度来决定下一站应该到哪里。

6】 接班的时候，旧调度不必告诉新调度最终目的应该怎么走，因为所有的调度都知道这个目的，一看就知道该走什么路了。

例如，有客户从新疆发货到青岛，由于新疆没有直达青岛的航班或者火车，所以只能先到达北京，然后再从北京直达青岛。

新疆的调度收到货物之后，他查找路由表，发现要到青岛，必须先到北京。新疆的调度会在货物上贴上青岛的标签而不是北京的标签，但是发货的时候，调度会选择将货物运送到新疆到北京的火车上。

货物到达北京之后，北京货运分公司的调度收到这件货物，首先查看这件货物的最终目的地址，然后北京调度也去查找路由表。他的路由表与新疆调度的路由表不同，在他的表上，北京到青岛有直达的火车，所以北京调度立即将货物原封不动的送上去青岛的火车。就这样一站一站的往前送(路由转发)，货物最终从新疆到达了青岛。



思考 那么调度是怎么知道全球地址表(路由表)的呢？这个表的生成是一个复杂的学习阶段，可以通过调度自行学习或者调度之间相互通告，也可以通过手工录入。前者称为动态路由，后者称为静态路由。

路由器充当的就是调度的角色。比如在青岛想访问一个位于北京的服务器，具体步骤如下。

1】 首先必须知道这个服务器的 IP 地址，然后用这个 IP 地址作为最终目的地址组装成数据包，发送给位于青岛的 Internet 提供商机房中的路由器。

2】 这个路由器收到这个包后，解析其目的 IP 地址，然后查找其路由表，发现这个目的 IP 地址的包应该从 1 号端口转发出去，所以它立即将这个包原封不动的向一号口转发。一号口通过光缆直接连接到了位于河北机房中的另一台路由器。



提示 当然青岛到河北之间不可能只用一条连续不断的光缆连接，中途肯定经过一些光缆通信中继站。

3】 河北的路由器收到这个 IP 包后，同样根据目的 IP 地址查找路由表，发现这个目的地址的包应该从 8 号端口中转发，它立即将这个包转发向 8 号端口。

4】 8 号端口通过光缆直接连接到了位于北京机房的一台路由器。

5】 这台路由器同样查找路由表做转发动作。

6】 经过一层层的寻找，最终找到了北京的这台服务器，将这个包传送到这台服务器的网卡，并提交到 TCP/IP 协议处理内存空间中。

7】 经过解析和处理，服务器发现最终的数据是一个 TCP 握手数据包，然后 TCP/IP 程序立刻返回一个确认包，再次返回给服务器一个确认包。三次握手完成后，就可以向服务器发送 HTTP 请求来获取它的网页资源了。

7.3.6 数据链路层

数据链路层就是指连通两个设备之间的链路，数据要经过这条链路来传递给对方。数

据链路层的程序将上层的数据包再次打包成对应链路的特定格式，按照对应链路的规则在链路上传输到对方。

数据链路就好比交通规则。在高速公路或者铁路上是需要遵守规则的，不能超速，不能乱停车，不能开车灯到最亮等。上路之前，先要看看公路的质量怎么样，是不是适合跑车或者先和对方商量一下传输的事宜。这就是链路层协商。

链路层的作用

首先是协商链路参数，比如双工、速率、链路质量等。

其次是将上层数据内容打包成帧，加上同步头进行传输，一次传输一句或者一个字符一个字符的传输(取决于上层的选择)。

最后，链路层程序调用物理层提供的接口，将帧提交给物理层。

相对于传输层的保障来说，OSI 的数据链路层也提供一些保障机制。比如一些链路层协议会在每个帧后面加一个校验字段，如果对方收到的帧的校验值与这个校验字段不符，则证明链路受到干扰，数据产生畸变，那么就将这一帧视为无效帧直接丢弃，不会向上层报告这个错误，因为上层对链路层的错误不关心。而接收方的传输层协议会感知某个包没有到达或者不完整，接收方的传输层协议会要求发送方重新传送这个不完整或者没有接收到的包，也就是端到端的保障传输。链路层只侦错，不纠错，而传输层既侦错，又纠错。

根据 OSI 模型，两台路由器或者交换机之间传送数据也属于两个系统间的互联，那么它们也一定遵循 OSI 的模型。下面就来分析一下，两台 PC 之间通信和两个路由器之间通信有什么区别。PC 间通信我们上文已经描述过，下面来讲一下路由间的通信。

简单的路由器设备工作在 OSI 的第三层，即网络层。它只处理下三层的内容，只有下三层的处理逻辑，而没有上 4 层的处理逻辑。路由器收到包后，只检查包中的 IP 地址，不会改变任何 IP 头之上的其他内容，最简单的路由器甚至不会改变 IP 头。在一些带有诸如 NAT 功能的路由器上，可能会对 IP 包的源或者目的 IP 地址做修改。数据包流入路由器后，路由器只分析到第三层的 IP 头，便可以根据路由表完成转发逻辑。

如图 7.1 所示为通信路径上各个设备所作用的层次示意图，具体如下。

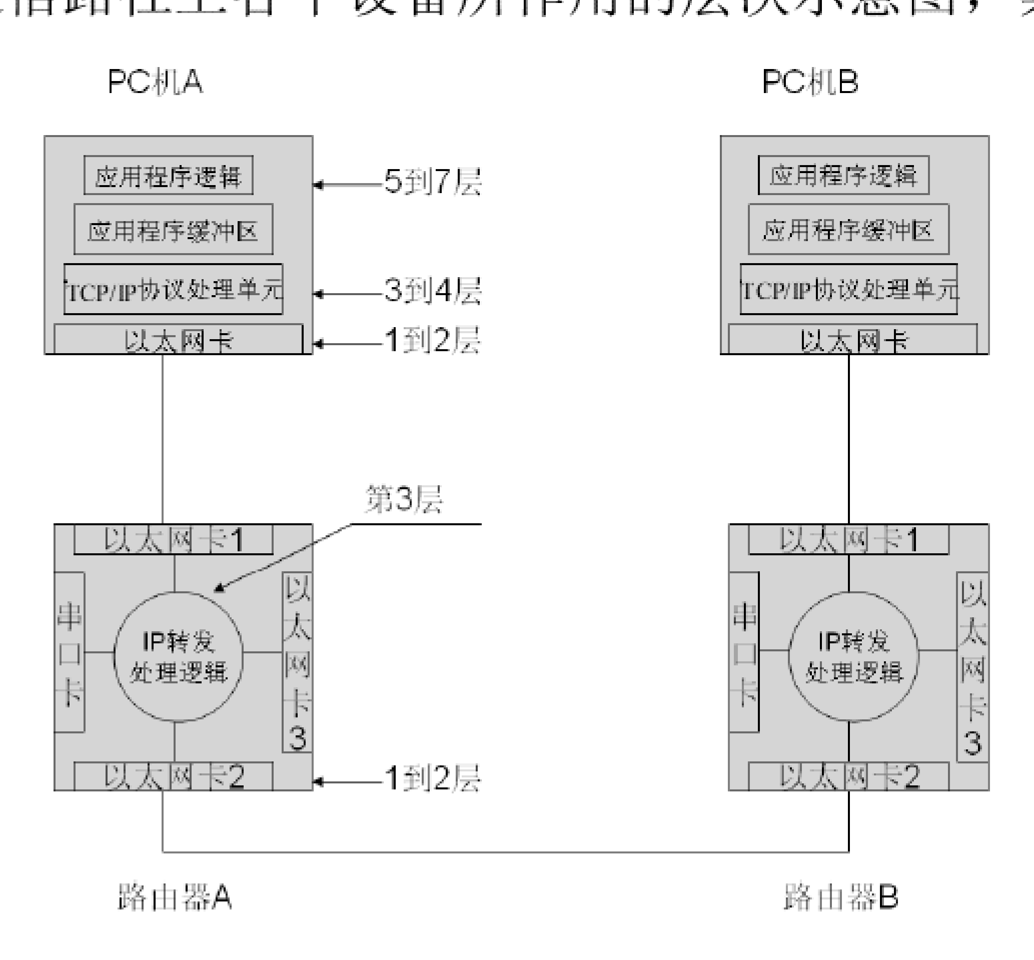


图7.1 通信路径上各个设备所作用的层次

1】 左边的 PC 机 A 连接到路由器 A 的以太网卡 1 上，路由器 A 的网卡 2 与路由器 B

的网卡 2 相连，右边的 PC 机 B 连接到路由器 B 的网卡 1 上。此时，要用 PC 机 A 上的 IE 浏览器访问位于 PC 机 B 上的 Web 服务，在 IE 浏览器的地址栏中输入 PC 机 B 的 IP 地址并回车后，IE 浏览器便会调用 WinSock 接口来访问操作系统内核的 TCP/IP 协议栈。IE 浏览器告诉 TCP/IP 协议栈它所访问的目的 IP 地址和目的端口，并把要发送的数据告诉 TCP/IP 协议栈。IE 浏览器发送给 PC 机 B 的数据，当然是一个 HTTP GET 请求，具体内容属于上三层，在这里不关心也不做分析。

2】 TCP/IP 协议栈收到这个数据之后，发现 IE 浏览器与 PC 机 B 当前并不存在连接，所以它首先要向 PC 机 B 上的 TCP/IP 协议栈发起连接请求，也就是 TCP 的三次握手过程。PC 机 A 的 TCP/IP 协议栈先组装第一次握手 IP 包，组好后发送给操作系统内核缓冲区，内核调用网卡驱动程序从缓冲区内将这个 IP 包编码并在线路上传递出去。握手数据包很小，只要一个以太网帧就可以容纳。

3】 这个帧最终到达路由器 A 的网卡 1 缓冲区内。网卡 1 产生中断信号，然后将这个帧去掉以太网头，发送到路由器 A 的内存中，等待 IP 转发逻辑模块的处理。运行在路由器 A 上的 IP 转发逻辑模块，其实就是 IP 路由协议计算模块，这个模块分析此 IP 包的头部目的 IP 地址，查找路由表以确定这个包将从哪个接口发送出去。IP 路由运算一定要快速高效，才不至于对网络性能造成瓶颈。

4】 路由器 A 查找路由表发现这个包应当从网卡 2 转发出去，所以它立即将这个包发送到网卡 2 并通过线路传送到了路由器 B 的网卡 2 上。经过同样的过程，路由器 B 将这个包路由到 PC 机 B 的网卡缓冲区内，PC 机 B 网卡产生中断，将这个包通过总线传送到 PC 机 B 的 TCP/IP 协议栈缓冲区内存。

5】 运行在 PC 机 B 上的 TCP/IP 协议栈程序分析这个包，发现 IP 是自己的， TCP 端口号为 80，握手标识位为二进制 1，就知道这个连接是由源地址 IP 所在的设备向自己的 80 端口，也就是 Web 服务程序所监听的端口发起的握手连接。根据这个逻辑，TCP/IP 协议栈返回握手确认 IP 包给 PC 机 A，PC 机 A 再返回一个最终确认包，这样就完成了 TCP 的三次握手。

6】 握手成功后，PC 机 A 上的 TCP/IP 协议栈立即将其缓冲区内将由 IE 浏览器发送过来的 HTTP GET 请求数据组装成 TCP/IP 数据包，发送给 PC 机 B。PC 机 B 得到这个数据包之后，分析其 TCP 端口号，并根据对应关系将数据放到监听这个端口的应用程序的缓冲区内存。

7】 应用程序收到这个 GET 请求之后，便会触发 Web 服务逻辑流程，返回 Web 网页数据，同样经由 PC 机 B 的 TCP/IP 协议栈，发送给 PC 机 A。

上述过程是一个正常通信的过程。



如果在 PC 机 B 向 PC 机 A 传送网页数据的时候，路由器 A 和路由器 B 之间的链路发生了几秒钟的短暂故障后又恢复连通性，这期间丢失了很多数据。虽然这样，依靠 TCP 协议的纠错功能，数据依然会被顺序的传送给 PC 机 A。

我们就来分析一下 TCP 是如何做到的。假如，在链路中断的时候，恰好有一个帧在链路上传送。发生故障后，这个帧就永久的丢失了。即使链路恢复后，路由器也不会重新传

送这个帧。但 PC 机 B 由于很长时间都没有收到 PC 机 A 的确认信息，便知道刚才发送的数据包可能已经被中途的网络设备丢弃了，所以 PC 机 B 上的 TCP 协议将重新发送这个数据包。



未接收到确认的包会存放在缓冲区内，不会删除，直到收到对方确认。

所以，即便中途经过的网络设备将这个包丢弃了，运行在通信路径最两端的 TCP/IP 协议，依然会重传这些丢弃的包，从而保障了数据传输，这也就是端到端的传输保障。只有端到端的保障，才是真正的保障，因为中途网络设备不会缓存发送的数据，更不会自动重传。

7.3.7 物理层

物理层就是研究在一种介质上(或者真空)如何将数据编码发送给对方。如果选择公路来跑汽车，要根据沥青路或者土路来选用不同的轮胎；如果选择利用空气来跑飞机，则需根据不同的气流密度来调整飞行参数，如果选择了真空，则只能利用电磁波或者光来传输，可以根据障碍物等因素选择不同波长的波来承载信号；如果选择了海水，则要根据不同的浪高来调整航海参数。这些都是物理层所关心的。

物理层和链路层

物理层和链路层是很容易混淆的两个层次。链路层是控制物理层的。物理层好比一个笨头笨脑的传送带，它不停地在运转，只要有东西放到传送带上就会被运输到对方。不管给它什么东西，它都一视同仁并且不会停下。

假设你我之间有一个不停运转的传送带，某时刻我有一大批货物要传送给你，是否可以一股脑的把这些货物不停地放到传送带上，一下子传送给你呢？当然可以，但是那样将没有整理货物的时间，永远处于不停地从传送带上拿下货物的状态，货物越堆越多，最终造成崩溃。如果能将货物一批一批的传送过来，不但给予了双方充足的整理货物的时间，而且使得货物传输显得井井有条。而将货物分批这件事，传送带本身是不会做的，只能靠 TCP 或者 IP 来做。链路层给每批货物附加上一些标志性的头部或者尾部，接收方看到这些标志，就知道一批货物又来了，并做接收动作。

每种链路，都有自己的一个最适分批大小，叫做最大传输单元，MTU。每次传输，链路上最大只能传输 MTU 大小的货物。如果要在一次传输中传送大于这个大小的货物，超过了链路接收方的处理吞吐量，则可能造成接收方缓冲区溢出或者强行截断等错误。

TCP 和 IP 这两个协议程序都会给货物分批。第一个分批的是 TCP，下到 IP 这一层，又会根据链路层的分批大小来将 TCP 已经分批的货物再次分批，如果 TCP 分批小于链路层分批，则 IP 不需要再分。如果大于链路层的分批，则 IP 会将货物分批成适合链路层分批的大小。被 IP 层分批的货物，最终会由接受方的 IP 层来再组装合并，但是由 TCP 分批的货物，接收方的 TCP 层不会合并，TCP 可以任意分割货物进行发送而接收的时候并不做合并的动作。对货物的处理分析全部交由上层应用程序来处理，所以利用 TCP/IP 通信的应用程序必须对自己所发送的数据有定界措施。

说白了，物理层就是用什么样的线缆、什么样的接口、什么样的物理层编码方式，归零还是不归零，同步方式，外同步还是内同步。高电压范围，低电压范围，电气规范等的东西。

通过物理层编码后，我们的数据最终变成了一串 bit 流，通过电路振荡传输给对方。对方收到 bit 流后，提交给链路层程序，由程序处理，剥去链路层同步头、帧头帧尾、控制字符等，然后提交给网络层处理程序(TCP/IP 协议栈等)，IP 头是个标签，收件人通过 IP 头来查看这个货物是谁发的。TCP 头在完成押运使命之后，还有一个作用就是确定由哪个上层应用程序来处理收到的包(用端口号来决定)。应用程序收到 TCP 提交的数据后，进行解析处理。

7.4 OSI 与网络

网络就是由众多节点通过某种方式互相连通之后所进行的多点通信系统。既然牵扯到节点与节点间的通信，那么就会符合 OSI 模型。

首先我们看看计算机总线网络。CPU、内存、外设三者通过总线互相连接起来，当然总线之间还有北桥和南桥，这两个芯片就犹如 IP 网络中的 IP 路由器或者网桥。CPU 与内存这两个部件都连接到北桥这个路由器上，然后北桥连接到南桥，南桥下连一个 HUB 总线，HUB 上连接了众多的外设，这些外设共享这个 HUB 与南桥进行通信。

> **提示** 说到 HUB，不要认为是专指以太网中的 HUB，HUB 的意思就是一条总线。如果在这条总线上运行以太网协议，则就是以太网 HUB，如果在这个 HUB 上运行的是 PCI 协议，则就是 PCI HUB(PCI 总线)。

连接到以太网 HUB 上的各个节点，采用 CSMA/CD 的竞争机制来获取总线使用权， PCI 总线同样采用仲裁竞争机制，只是实现方式不同。实现方式也可以称其为协议，所以有以太网 HUB 和 PCI HUB 之分，也就是说 HUB 上运行的是不同的协议。当然以太网 HUB 设计要求远远比 PCI HUB 低，速度也低很多。

图 7.2 所示的模型是一个常见的小型网络，几台 PC 机通过以太网 HUB 和路由器互相连接起来，然后通过运行在每台 PC 机上的 TCP/IP 协议来通信。路由器的作用只是分析目的 IP 地址从而做转发动作。

而我们再观察一下图 7.3。发现除了连接各个组件之间的线路变成了并行多线路之外，其他没有什么大的变化。但是，这两个网络的通信过程，是有区别的。上面的网络利用一种高级复杂的协议——TCP/IP 协议来通信，而图 7.3 所示的网络是通过直接总线协议进行通信。在下面的网络中，各个部件之间的连线非常短，速度很高且非常稳定，自身就可以保障数据的稳定传输，所以不需要 TCP 这种传输保障协议的参与。在上面的网络中，各个部件之间可能相隔很远的距离，链路速度慢，稳定性不如主板上的导线高，所以必须运行一种端到端的传输保障协议，比如 TCP 协议，来保障端到端的数据传输。

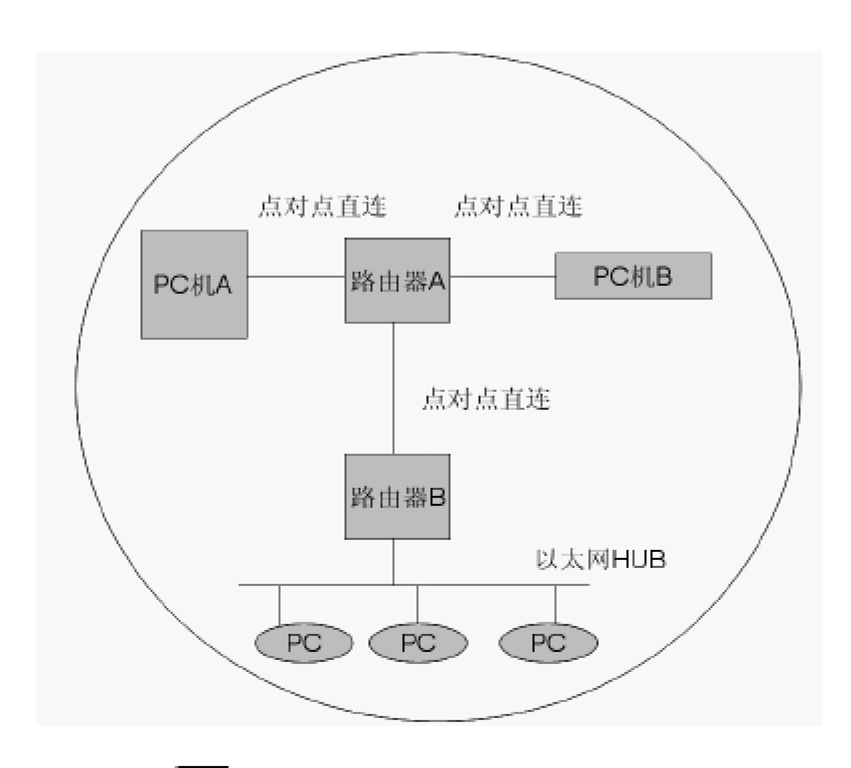


图7.2 一个典型的网络

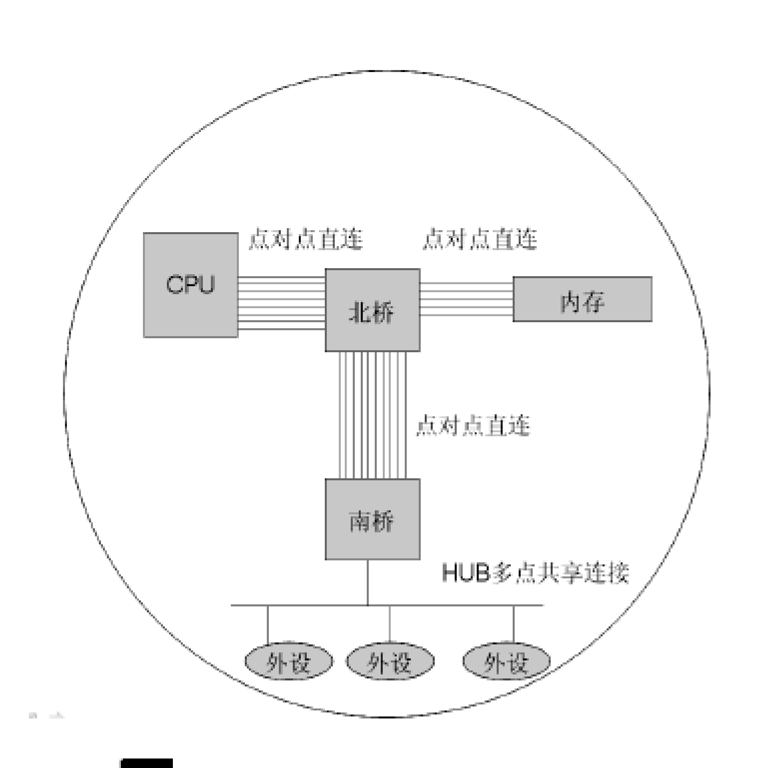


图7.3 计算机总线网络

此外，上面的两个网络模型，其本质是相同的，因为它们两个都是从基本原理发展而来的。我们说，这两个网络模型都符合 OSI 这个抽象模型。再甚者，这两个模型都符合“连、找、发”抽象模型。

- 首先，所有部件之间都用了导线来连接。对于第一个模型，导线为双绞线或者其他形式的外部电缆；对于第二个模型，导线为电路板上印刷的蛇行线。这就是所谓“连”。
- 其次，这两个模型中都有寻址的逻辑。第一个模型利用 IP 地址作为寻址方法；第二个模型中利用地址总线作为寻址方法。这就是所谓“找”。

> **提示** 生物细胞之间的通信，同样符合 OSI 模型和“连找发”模型。细胞之间通过血管来传递信息，这就是“连”；通过配体－受体关系来找到目标，这就是“找”；血液流动将配体分子传递(广播)到人体的每个角落，这就是“发”。我国分子生态学创始人向近敏曾经提出分子信息网络学说，就恰恰体现了网络的思想。在分子上层，还有细胞信息网络学说和遗传信息网络学说，它们一个比一个高层，一个比一个抽象。然而分子信息网络也不一定就是最底层的网络，或许还有原子信息网络，电子信息网络等。

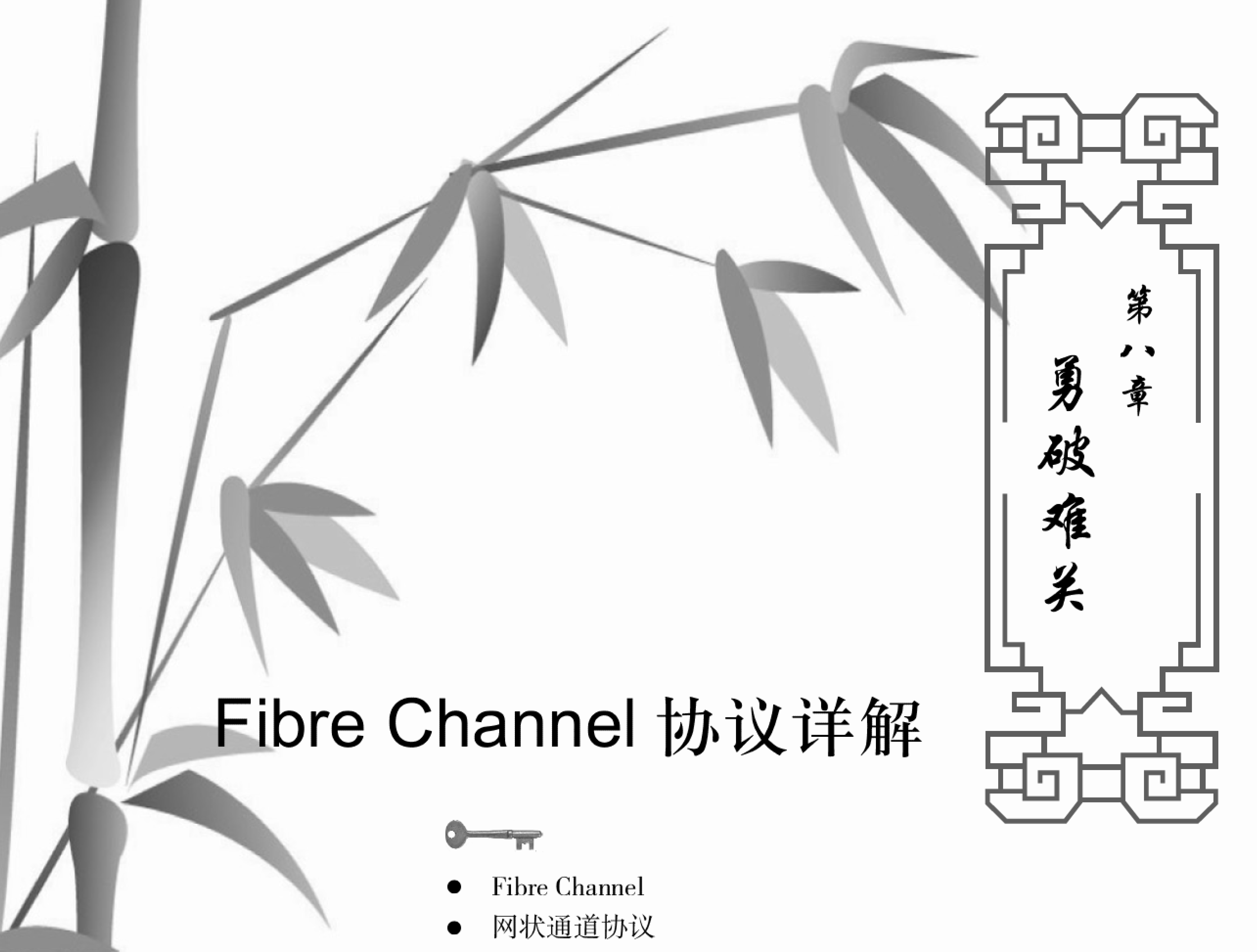
- 最后，第一个模型利用 TCP 协议进行有保障的数据发送动作，第二个模型中由于线路非常稳定，不需要高级协议参与，而是直接利用电路逻辑从目标部件将数据复制过来。这就是所谓“发”。

网中有网

我们在以前的章节中，多次提到过“网中有网”这个词。而我们现在再来体会一下，发现计算机系统、计算机网络、Internet，这些系统，确实可以用网中有网来描述。计算机总线这个微型网络，通过一个网卡，接入以太网交换机或者 HUB，与其他计算机总线网络形成一个局域网，然后这个局域网再连接到路由器网关，从而连接到更大的网络，甚至 Internet。

所有的网络，都按照 OSI 和“连找发”模型有条不紊的通信交互着，为我们服务。分子之间和细胞之间神奇的相互作用着，地球和月球有条不紊的旋转运行着，太阳系缓慢的自转，并围绕着更大的银河系旋转。

读 书 笔 记



Fibre Channel 协议详解

- Fibre Channel
- 网状通道协议
- 光纤通道协议
- OSI

本书的第 6 章末尾，引出了 SAN 的概念。SAN 首先是个网络，而不是指存储设备。当然，这个网络是专门用来给主机连接存储设备用的。这个网络中有着很多的元件，它们的作用都是为了让主机更好的访问存储设备。

SAN 概念的出现，只是个开头而已，因为按照 SCSI 总线 16 个节点的限制，不可能接入很多的磁盘。要扩大这个 SAN 的规模，还有很长一段路要走。如果仅仅用并行 SCSI 总线，那么 SAN 只能像 PCI 总线一样作为主机的附属品，而不可能成为一个真正独立的“网络”。必须找到一种可寻址容量大、稳定性强、速度快、传输距离远的网络结构，从而连接控制器和磁盘或只有连接控制器到主机。

干脆破釜沉舟，独立研发一套全新的网络传输系统，专门针对局部范围的高速高效传输。

然而，形成一套完整的网络系统并非易事，首先必须得有个蓝图。这个蓝图是否有现成可以参考的呢？当然有，OSI 就是一个经典的蓝图。OSI 是对任何互联系统的抽象。

8.1 FC 网络——极佳的候选角色

FC 协议自从 1988 年出现以来，已经发展成为一项非常复杂、高速的网络技术。它最初并不是研究来作为一种存储网络技术的。最早版本的 FC 协议是一种为了包括 IP 数据网在内的多种目的而推出的高速骨干网技术，它是作为惠普、Sun 和 IBM 等公司组成的 R&D 实验室中的一项研究项目开始出现的。曾经有几年，FC 协议的开发者认为这项技术有一天会取代 100BaseT 以太网和 FDDI 网络。在 20 世纪 90 年代中期，还可以看到研究人员关于 FC 技术的论文。这些论文论述了 FC 协议作为一种高速骨干网络技术的优点和能力，而把存储作为不重要的应用放在了第二位。

Fibre Channel 也就是“网状通道”的意思，简称 FC。

> **提示** 由于 Fiber 和 Fibre 只有一字之差，所以产生了很多流传的误解。FC 只代表 Fibre Channel，而不是 Fiber Channel，后者被翻译为“光纤通道”，甚至接口为 FC 的磁盘也被称为“光纤磁盘”，其实这些都是很滑稽的误解。

不过到目前为止，似乎称 FC 为光纤而不是直接称其 FC 的文章和资料更多。这种误解使得初入存储行业的人摸不着头脑，认为 FC 就是使用光纤的网络，甚至将 FC 与使用光纤传输的以太网链路混淆起来。在本书内不会使用“光纤通道”或者“光纤磁盘”这种定义，而统统使用 FC 和“FC 磁盘”。相信在阅读完本章之后，大家就不会再混淆这些概念了，会知道 FC 与光纤根本就没有必然地联系。

Fibre Channel 可以称其为 FC 协议，或 FC 网络、FC 互联。像 TCP/IP 一样，FC 协议集同样具备 TCP/IP 协议集以及以太网中的很多概念，比如 FC 交换、FC 交换机、FC 路由、FC 路由器，SPF 路由算法等。我们完全可以类比地看待 TCP/IP 协议以及 FC 协议，因为它们都遵循 OSI 的模型。任何互联系统都逃不过 OSI 模型，不可能存在某种不能归属于 OSI 中某个层次的元素。

下面我们用 OSI 来将 FC 协议进行断层分析。

8.1.1 物理层

OSI 的第一层就是物理层。作为一种高速的网络传输技术，FC 协议体系的物理层具有比较高的速度，从 1Gb/s、2Gb/s、4Gb/s 到当前的 8Gb/s。作为高速网络的代表，其底层也使用了同步串行传输方式，而且为了保证传输过程中的电直流平衡、时钟恢复和纠错等特性，其传输编码方式采用 NMb 编码方式。

为了实现远距离传输，传输介质起码要支持光纤。铜线也可以，但是距离受限制。FC 协议集中物理层的电气子层名为 FC0，编码子层名为 FC1。

8.1.2 链路层

1. 字符编码以及 FC 帧结构

现代通信在链路层一般都是成帧的，也就是将上层发来的一定数量的位流打包加头尾传输。FC 协议在链路层也是成帧的。既然需要成帧，那么一定要定义帧控制字符。

FC 协议定义了一系列的帧控制策略及对应的字符。这些控制字符不是 ASCII 码字符集中定义的那些控制字符，而是单独定义了一套专门用于 FC 协议的字符集，称为“有序集”。其中的每个控制字符其实是由 4 个 8 位字节组成的，称为一个“字”(word)，而每个控制字开头的一个字节总是经过 8 10b 编码之后的 0011111010(左旋)或者 1100000101(右旋)。

由于还没有标准名词出现，所以不得不引入“左旋”和“右旋”这两个化学名词来描述这种镜像编码方式。左旋和右旋是指 1 和 0 对调。编码电路可以根据上一个 10 位中所包含的 1 的个数来选择下一个 10 位中 1 的个数。如果上一个 1 的个数比 0 的个数少，那么下一个 10 位中就编码成 1 的个数比 0 的个数多，这样总体平衡了 1 和 0 的个数。

0011111010 左旋或者 1100000101 右旋，FC 协议给这个字符起了一个名字，叫做 K28.5。这个字未经过 8 10b 编码之前的值是十六进制 BC，即 10111100，它的低 5 位为 11100(十进制的 28)，高 3 位为 101(十进制的 5)。FC 协议便对这个字表示为“K28.5”，也就是说高三位的十进制是 5，低 5 位的十进制是 28，这样便可以组合成相应的二进制位码。然后再加上一个描述符号 K(控制字符)或者 D(数据字符)。K28.5 这个字符没有 ASCII 字符编码与其冲突，它的二进制流中又包含了连续的 5 个 1，非常容易被电路识别，当然符合这些条件的字符还有好几个。

每个控制字均由 K28.5 字符开头，后接 3 个其他字符(可以是数据字符)，由这 4 个字符组成的字来代表一种意义，比如 SOF(Start Of Frame)、EOF(End Of Frame)等。

定义了相关的控制字之后，需要定义一个帧头了。FC 协议定义了一个 24 字节的帧头。以太网帧头才 14 字节，用起来还绰绰有余，为什么 FC 需要定义 24 字节呢？在这个问题上，协议的设计者独树一帜，因为这 24 字节的帧头不但包含了寻址功能，而且包含了传输保障的功能。网络层和传输层的逻辑都用这 24 字节的信息来传递。

我们知道，基于以太网的 TCP/IP 网络，它的开销一共是：14 字节(以太网帧头)+20 字节(IP 头)+20 字节(TCP 头)=54 字节，或者把 TCP 头换成 8 字节的 UDP 头一共是 42 字节。这就注定了 FC 的开销比以太网加上 TCP/IP 的开销要小，而实现的功能都差不多。

可以看出来，以太网中用于寻址的开销太大，一个以太网 MAC 头和一个 IP 头这两个就是已经 34 字节了，更别说再加上 TCP 头了。而 FC 将寻址、传输保障合并起来放到一个头中，长度才 24 字节。图 8.1 所示的是一个 FC 帧的示意图，图 8.2 是一个 FC 帧编码之后的表示。

2. 链路层流量控制

在链路层上，FC 定义了两种流控策略，一种为端到端的流控，另一种为缓存到缓存的流控。端到端流控比缓存到缓存流控要上层和高级。在一条链路的两端，首先面对链路的一个部件就是缓存。接收电路将一帧成功接收后，就放入了缓存中。如果由于上位程序处理缓慢而造成缓存已经充满，FC 协议还有机制来通知发送方减缓发送。如果链路的一端是

FC 终端设备，另一端是 FC 交换机，则二者之间的缓存到缓存的流量控制只能控制这个 FC 终端到 FC 交换机之间的流量。

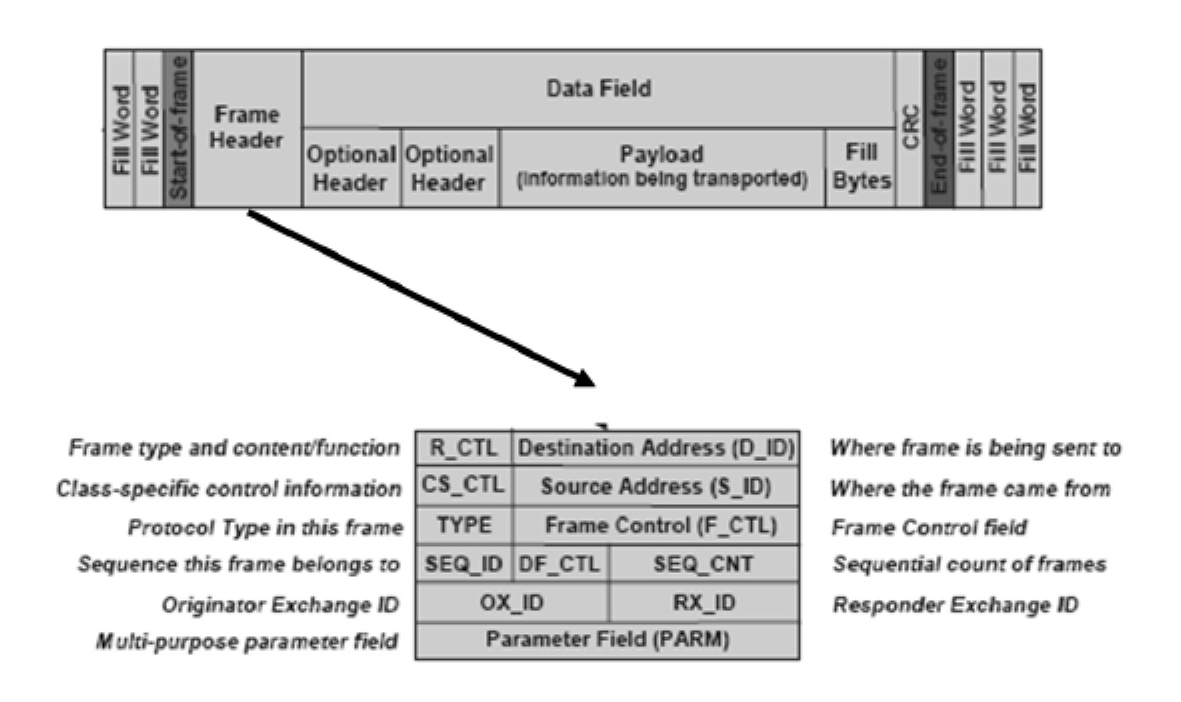


图8.1　一个 FC 帧的结构

```
000001: K28.5 D21.4 D21.5 D21.5  <---- IDLE Ordered Set
000002: K28.5 D21.4 D21.5 D21.5  <---- IDLE Ordered Set
000003: K28.5 D21.5 D22.2 D22.2  <---- Start-of-Frame Delimiter
000004: D05.0 D00.0 D00.0 D02.0  <---- Frame Header (Word 0)
000005: D00.0 D00.0 D00.0 D04.0  <---- Frame Header (Word 1)
000006: D08.0 D09.4 D00.0 D00.0  <---- Frame Header (Word 2)
000007: D31.7 D00.0 D00.0 D00.0  <---- Frame Header (Word 3)
000008: D10.1 D10.1 D31.7 D31.7  <---- Frame Header (Word 4)
000009: D00.0 D00.0 D00.0 D00.0  <---- Frame Header (Word 5)
00000A: D00.0 D00.0 D00.0 D00.0  <---- Payload (Word 0)
00000B: D00.0 D00.0 D02.0 D00.0  <---- Payload (Word 1)
00000C: D00.0 D00.0 D00.0 D00.0  <---- Payload (Word 2)
00000D: D08.2 D17.2 D17.1 D25.2  <---- Cyclic Redundancy Check (CRC)
00000E: K28.5 D21.5 D21.3 D21.3  <---- End-of-Frame Delimiter
00000F: K28.5 D21.4 D21.5 D21.5  <---- IDLE Ordered Set
000010: K28.5 D21.4 D21.5 D21.5  <---- IDLE Ordered Set
000011: K28.5 D21.4 D10.2 D10.2  <---- R_RDY Ordered Set
000012: K28.5 D21.4 D21.5 D21.5  <---- IDLE Ordered Set
```

图8.2　一个完整的 FC 帧的有序集表示

而通信的最终目的是网络上的另一个 FC 终端，这之间可能经历了多个 FC 交换机和多条链路。而如果数据流在另外一个 FC 终端之上发生拥塞，则这个 FC 终端就必须通知发起端降低发送频率，这就是 “端到端”的流量控制。图 8.3 表示了这两种机制的不同。

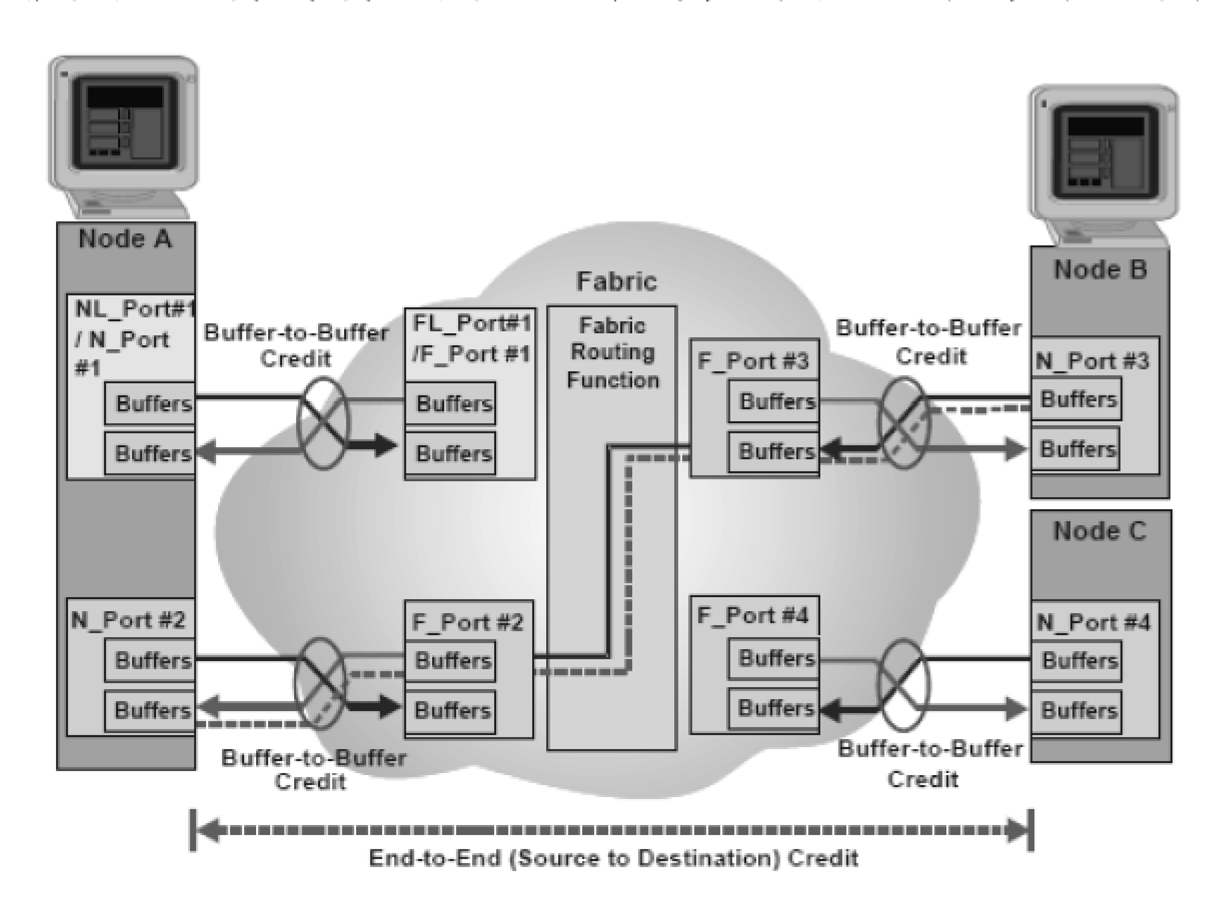


图8.3　B2B 和 E2E 两种方式的流量控制示意图

3. MTU

一般情况下，以太网的 MTU 为 1500 字节，而 FC 链路层的 MTU 可以到 2112 字节。这样，FC 链路层相对以太网链路层的效率又提高了。

8.1.3　网络层

1. 拓扑

同以太网类似，FC 也提供了两种网络拓扑模式：FC-AL 和 Fabric。

- FC-AL

  FC-AL 拓扑类似于以太网共享总线拓扑，但是连接方式不是总线，而是一条仲裁环路(Arbitral Loop)。每个 FC AL 设备首尾相接构成了一个环路。一个环路能接入的

最多节点是 128 个，实际上是用了一个字节的寻址容量，但是只用到了这个字节经过 8 10 b 编码之后奇偶平衡(0 和 1 的个数相等)的值，也就是 256 个值中的 134 个值来寻址，这些被筛选出来的地址中又被广播地址、专用地址等占用了，最后只剩下 127 个实际可用的节点地址。

图 8.4 为 4 个 FC-AL 设备接入一个仲裁环的拓扑图。仲裁环是一个由所有设备“串联”形成的闭合环路。如果某个设备发生故障，这个串联的环路是不是就会全部瘫痪呢？在 FC-AL 集线设备的每个接口上都有一套“旁路电路(Bypass Circuit)”，这套电路一旦检测到本地设备故障或电源断开，就会自动将这个接口短路，从而使得整个环路将这个故障的设备 Bypass 掉，不影响其他设备的工作。

数据帧在仲裁环内是一跳一跳被传输的，并且任何时刻数据帧只能按照一个方向向下游传输。图 8.5 为 AL 环路数据帧传输机制的示意图。

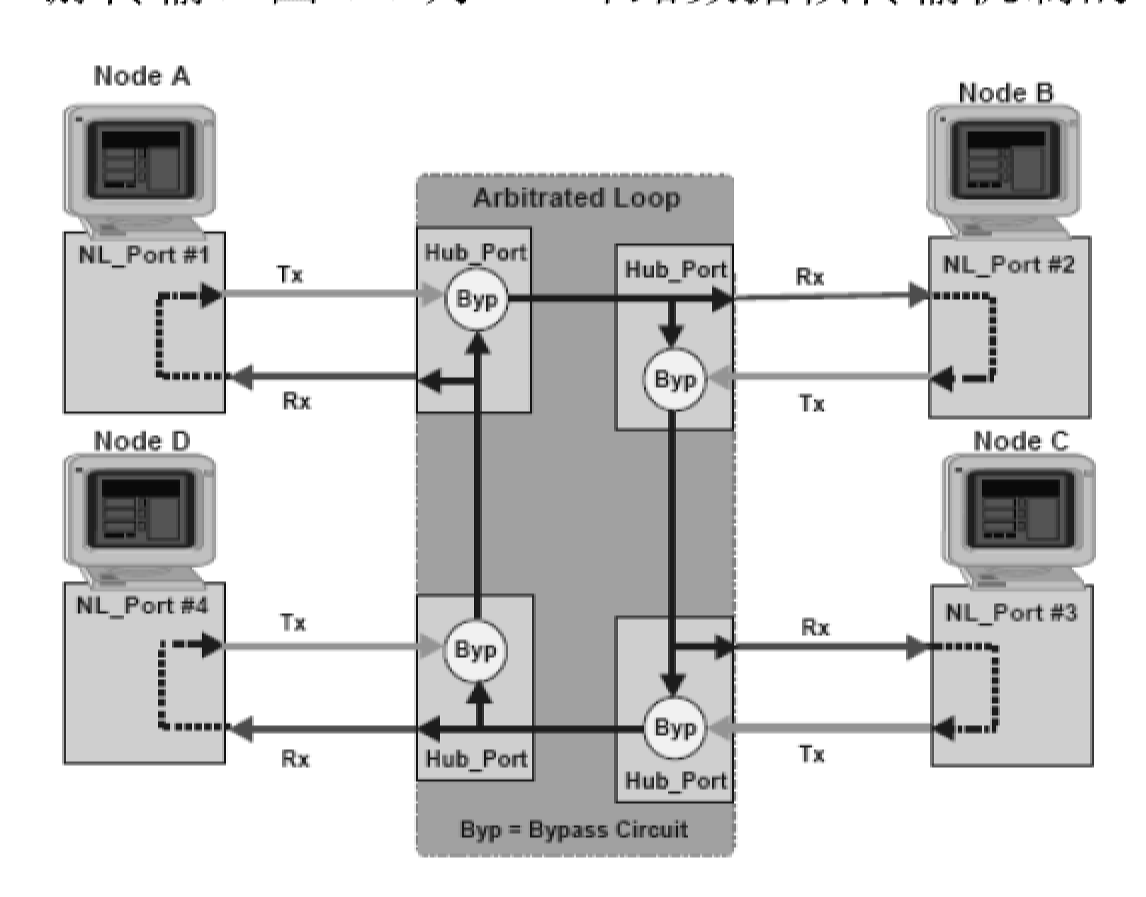


图8.4 FC 仲裁环结构示意图

图8.5 AL 环路数据传输机制示意图

在图 8.5 所示的仲裁环中，若 a 节点想与 h 节点通信，在 a 节点赢得仲裁之后，便向 h 节点发送数据帧。然而，由于这个环的数据是顺时针方向传递的，所以 a 发出的数据帧，只能先被 b 节点收到，由 b 节点接着传递到 c 节点，依次传递，最终传递到 h 节点。所以，虽然 a 和 h 节点之间只有一跳的距离，但是仍然需要绕一圈来传递数据。

- Fabric

另一种 Fabric 拓扑和以太网交换拓扑类似。Fabric 的意思为“网状构造”，表明这种拓扑其实是一个网状交换矩阵。

交换矩阵的架构相对于仲裁环路来说，其转发效率大大提高了，联入这个矩阵的所有节点之间都可以同时进行点对点通信，加上包交换方式所带来的并发和资源充分利用的特性，使得交换架构获得的总带宽为所有端口带宽之和。而 AL 架构下，接入环路的节点不管有多少，其带宽总为恒定，即共享的环路带宽。

图 8.6 为一个交换矩阵的示意图。每个 FC 终端设备都接入了这个矩阵的端点，一个设备发给另一个设备的数据帧被交换矩阵收到后，矩阵便会“拨动”这张矩阵网交叉处的开关，以连通电路，传输数据。可以将这个矩阵想象成一个大的电路开关矩阵，矩阵根据通信的源和目的决定拨动哪些开关。这种矩阵被做成芯片集

成到专门的交换机上，然后辅以实现 FC 逻辑的其他芯片或 CPU、ROM，就形成了一台用于 Fabric 交换的交换机。

图 8.7 所示的是一台 Fabric 交换机。FC 设备通过光纤或者铜线等各种标注的线缆连接到这台交换机上，便可以实现各个节点基于 FC Fabric 拓扑方式的点对点通信。

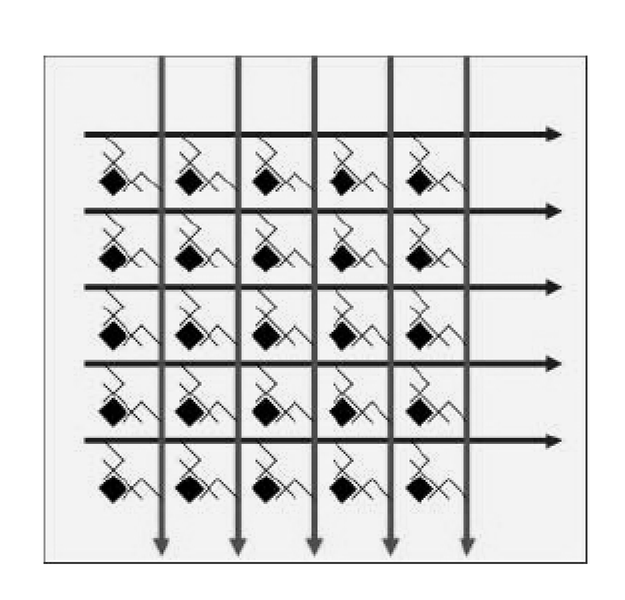


图8.6 Cross Bar 交换矩阵示意图



图8.7 Brocade 公司的 FC 交换机

FC 交换拓扑寻址容量是 2 的 24 次方个地址，比以太网理论值(2 的 48 次方)少。即便是这样，对于专用的存储网络来说也足够了，毕竟 FC 设计的初衷是用于存储网络的一种高速高效网络。

2. 寻址

任何网络都需要寻址机制，FC 当然也不例外了。

首先，像以太网端口 MAC 地址一样，FC 网络中的每个设备自身都有一个 WWNN(World Wide Node Name)，不管这个设备上有多少个 FC 端口，设备始终拥有一个固定的 WWNN 来代表它自身。然后，FC 设备的每个端口都有一个 WWPN(World Wide Port Name，世界范围的名字)地址，也就是说这个地址在世界范围内是惟一的，世界上没有两个接口地址是相同的。

FC Fabric 拓扑在寻址和编址方面与以太网又有所不同。具体体现在以太网交换设备上的端口不需要有 MAC 地址，而 FC 交换机上的端口都有自己的 WWPN 地址。这是因为 FC 交换机要做的工作比以太网交换机多，许多智能和 FC 的逻辑都被集成在 FC 交换机上，而以太网的逻辑相对就简单了许多，因为上层逻辑都被交给诸如 TCP/IP 这样的上层协议实现了。然而 FC 的 Fabric 网中，FC 交换机担当了很重要的角色，它需要处理到 FC 协议的最上层。每个 FC 终端设备除了和最终通信的目标有交互之外，还需要和 FC 交换机打好交道。

- WWNN

  每个 FC 设备都被赋予一个 WWNN，这个 WWNN 一般被写入设备的 ROM 中不能改变，但是在某些条件下也可以通过运行在设备上的程序动态的改变。

- WWPN 和三个 ID

  WWPN 地址的长度是 64 位，比以太网的 MAC 地址还要长出 16 位来。可见 FC 协议很有信心，认为 FC 会像以太网一样普及，全球会产生 2 的 64 次方个 FC 接口。然而，如果 8 个字节长度的地址用于高效路由的话，无疑是梦魇(IPv6 地址长度为 128 位，但是鉴于 Internet 的庞大，也只好牺牲速度换容量了)。所以 FC 协议决定在 WWPN 之上再映射一层寻址机制，就是像 MAC 和 IP 的映射一样，给每个连接到 FC 网络中的接口分配一个 Fabric ID，用这个 ID 而不是 WWPN 来嵌入链路帧

中做路由。这个 ID 长 24 位，高 8 位被定义成 Domain 区分符，中 8 位被定义为 Area 区分符，低 8 位定义为 PORT 区分符。

这样，WWPN 被映射到 Fabric ID，一个 Fabric ID 所有 24bit 又被分成 Domain ID、Area ID、Port ID 这三个亚寻址单元。

- Domain ID：用来区分一个由众多交换机组成的大的 FC 网络中每个 FC 交换机本身。一个交换机上所有接口的 Fabric ID 都具有相同的高 8 位，即 Domain ID。Domain ID 同时也用来区分这个交换机本身，一个 Fabric 中的所有交换机拥有不同的 Domain ID。一个多交换机组成的 Fabric 中，Domain ID 是自动被主交换机分配给各个交换机的。根据 WWNN 号和一系列的选举帧的传送，WWNN 最小者获胜成为主交换机，然后这个主交换机向所有其他交换机分配 Domain ID，这个过程其实就是一系列的特殊帧的传送、解析和判断。
- Area ID：用来区分同一台交换机上的不同端口组，比如 1、2、3、4 端口属于 Area 1，5、6、7、8 端口属于 Area2 等。其实 Area ID 这一层亚寻址单元意义不是很大。我们知道，每个 FC 接口都会对应一块用来管理它的芯片，然而每个这样的芯片却可以管理多个 FC 端口。所以如果一片芯片可以管理 1、2、3、4 号 FC 端口，那么这个芯片就可以属于一个 Area，这也是 Area 的物理解释。同样，在主机端的 FC 适配卡上，一般也都是用一块芯片来管理多个 FC 接口的。
- Port ID：用来区分一个同 Area 中的不同 Port。

经过这样的 3 段式寻址体系，可以区分一个大 Fabric 中的每个交换机、交换机中的每个端口组及每个端口组中的端口。

3. 寻址过程

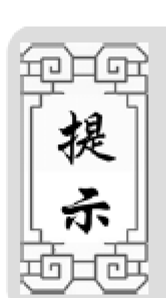
1) 地址映射

既然定义了两套编址体系，那么一定要有映射机制，就像 ARP 协议一样。FC 协议中地址映射步骤如下。

1】 当一个接口连接到 FC 网络中时，如果是 Fabric 架构，那么这个接口会发起一个登录注册到 Fabric 网络的动作，也就是向目的 Fabric ID 地址 FFFFFE 发送一个登录帧，称为 FLOGIN。

2】 交换机收到地址为 FFFFFE 的帧之后，会动态的给这个接口分配一个 24 位的 Fabric ID，并记录这个接口对应的 WWPN，做好映射。

3】 **此后这个接口发出的帧中不会携带其 WWPN，而是携带其被分配的 ID 作为源地址。**



以太网是既携带 MAC 地址，又携带 IP 地址，在效率上打了折扣。

4】 **如果接口是连接到 FC 仲裁环网络中，那么整个环路上的节点会选出一个临时节点(根据 WWPN 号的数值，最小的优先级最高)，然后由这个节点发送一系列的初始**

化帧，给每个节点分配环路 ID。



FC 网络中的 FCID 都是动态的，每个设备每次登录到 Fabric 所获得的 ID 可能不一样。同样，FC 交换机维护的 Fabric ID 与 WWPN 的映射也是动态的。

图 8.8 所示的是 FC 设备登录到 Fabric 过程示意图。

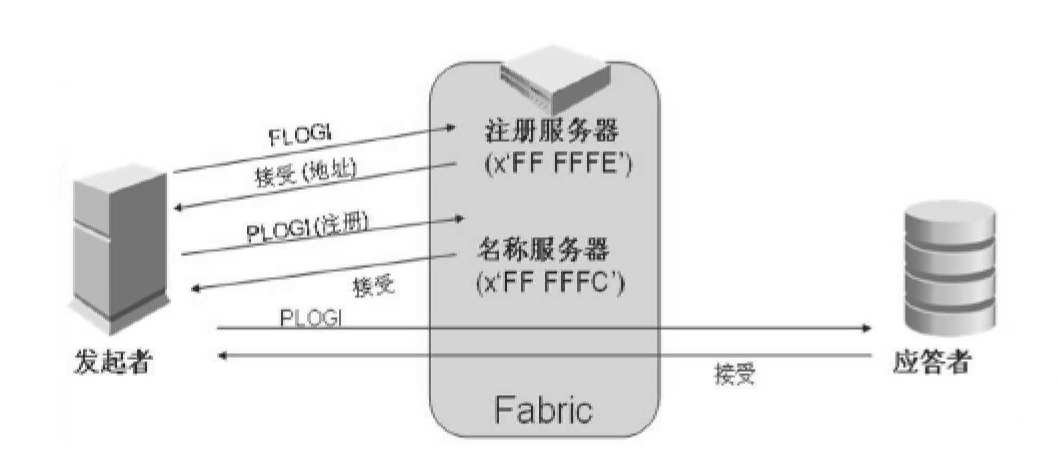


图8.8　FC 设备登录 Fabric 网络的过程

2)　寻址机制

编址之后就要寻址，寻址就牵扯到路由的概念。

一个大的 FC 网络中，一般有多台交换机相互连接，它们可以链式级联，也可以两两连接，甚至任意连接，就像 IP 网络中的路由器连接一样，但是 FC 网络不需要太多的人工介入。如果将几台交换机连接成一个 FC 网络，则它们会自动的协商自己的 Domain ID，这个过程是通过选举出一个 WWPN 号最小的交换机来充当主交换机，由主交换机来向下给每个交换机分配 Domain ID，以确保不会冲突。

对于寻址过程，这些交换机上会运行相应的路由协议。最广泛使用的路由协议就是 SPF(最短路径优先)协议，是一种很健壮的路由协议。比如用于 IP 网络中的 OSPF 协议，FC 网络也应用了这种协议。这样就可以寻址各个节点，进行各个节点无障碍的通信。

IP 网络需要很强的人为介入性，比如给每个节点配置 IP 地址，给每个路由器配置路由信息及 IP 地址等，这样出错率会很高。FC 网络中自动分配和管理各种地址，避免了人算带来的错误。FC 采用自动分配地址的策略，一个最根本的原因是 FC 从一开始就被设计为一个专用、高效、高速的网络，而不是给 Internet 用的，所以自动分配地址当然适合它。如果给 Internet 也自动分配地址，那么后果不堪设想。

既然要与目的节点通信，怎么知道要通信的目标地址是多少呢？我们知道，FC 被设计为一个专用网络，一个小范围、高效、高速、简易配置的网络。所以使用它的时候也非常简便，就像 Windows 中浏览网上邻居一样。

每个节点在登录到 FC 网络并且被分配 ID 之后，会进行一个名称注册过程，也就是接口上的设备会向一个特定的目的 ID 发一系列的注册帧，来注册自己。这个 ID 实际上并没有物理设备与其对应，只是运行在交换机上的一套名称服务程序而已，而对于终端 FC 设备来说，会认为自己是在和一个真实的 FC 设备通信。对于 Windows 系统来说，每台机器启动之后，如果设置了 WINS 服务器，会向 WINS 服务器来注册自己的主机名和 IP 地址。

每台机器都这么做，所以网络中的 WINS 服务器就会掌握网络中的所有机器的主机和

IP。同样，FC 交换机上运行的这个名称服务程序，就相当于 WINS 服务器。但是其地址是惟一的、特定的，不像 WINS 服务器可以被配置为任何 IP 地址。也就是说在 FC 协议中，这个地址是大家都公认不会去改变的，每个节点都知道这个地址，所以都能找到名称服务器。其实不是物理的服务器，只是运行在 FC 交换机上的程序，也可以认为 FC 交换机本身就是这台服务器。

节点注册到名称服务之后，服务便会将网络上存在的其他节点信息告诉这个接口上所连接的设备，就像浏览网上邻居一样，所以这个接口上的设备便知道了网络上的所有节点和资源。

- ZONE

  为了安全性考虑，可以进行人为配置，让名称服务器只告诉某个设备特定的节点。比如网络上存在 a、b、c、d 四个节点，可以让名称服务只向 a 通告 b、c 两个节点的存在，而隐藏 d 节点，这样 a 看不到 d。但是这样做有时候会显得很不保险，因为 a 虽然没有通过名称服务得到 d 的 ID，但是如果将节点 d 的 ID 直接告诉节点 a 的话，那么它就可以和 d 主动发起通信。而这一切，交换机不做干涉，因为交换机傻傻的认为只要名称服务器没有向 a 通告 d 的 ID，a 就不会和 d 发起通信。

  发生这种结果的原因是在物理上节点 a 和节点 d 并没有被分开，a 和 d 总有办法通信。就像有时网上邻居里看不到一台机器，但是它明明在线，那么如果此时知道那台机器的地址，照样可以不通过网上邻居，直接和它通信。如果两个节点被物理隔开了，那么就真的无能为力了。前者实现隔离的方法叫做软 ZONE，后者做法叫做硬 ZONE。

  所谓 ZONE，即分区的意思，同一个分区内的节点之间可以相互通信，不同分区之间的节点无法通信。软 ZONE 假设大家都是守法公民，名称服务器没有通告的 ID 就不去连接。而硬 ZONE 不管是否守法都会从底层硬件上强制隔离，即使某个节点知道了另外分区中某个节点的 ID，也无法和对方建立通信，因为底层已经被阻断了。图 8.9 是一个 Fabric ZONE 的示意图。

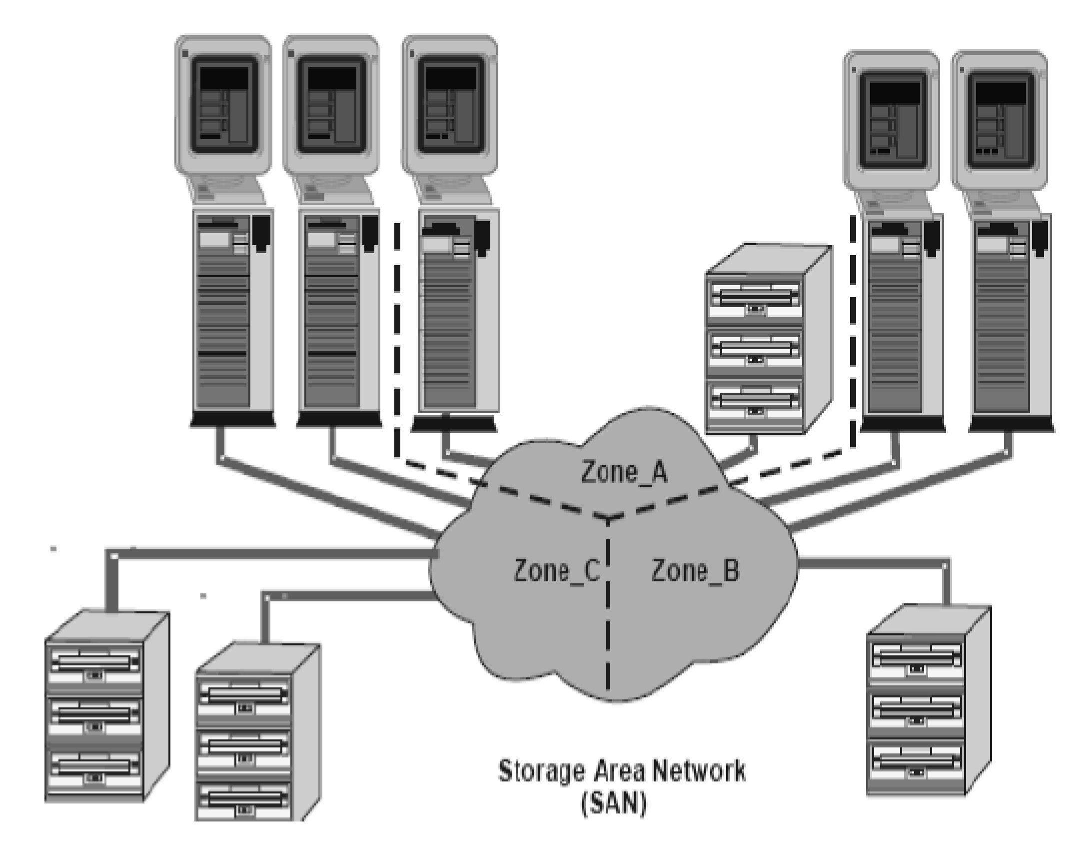


图8.9 一个具有三个 ZONE 的 Fabric

- 与目标通信

  从名称服务器得知网络上的节点 ID 之后，如果想发起和一个节点的通信，那么这个设备需要直接向目的端口发起一个 N_PORT Login 过程交换一系列的参数，然后

进行 Process Login 过程(类似于 TCP 向特定端口发送握手包一样)，即进行应用程序间的通信。这些 Login 过程其实就是上三层的内容，属于会话层，和网络传输已经没什么关联了。这些 Login 的帧也必须经过 FC 下 4 层来封装并传输到目的地，就像 TCP 握手过程一样。

名称服务器只是 FC 提供的所有服务中的一个，其他还有时间服务，别名服务等，这些地址都是事先定死的。

Fabric 网络中还有一种 FC Control Service，如果节点向这个服务注册，也就是向地址 FFFFFD 发送一个 State Change Registration (SCR)帧，那么一旦整个 Fabric 有什么变动，比如一个节点离线了、或者一个节点上线了、或者一个 ZONE 被创建了等，Fabric 便会将这些事件封装到 Registered State Change Notification (RSCN)帧里发送给注册了这项服务的所有节点。这个动作就像预订新闻一样，通常一旦节点被通知有这些事件发生之后，节点需要重新进行名称注册，以便从名称服务器得到网络上的最新资源情况，也就是刷新一下。

这些众所周知的服务都是运行在交换机内部的，而不是物理上的一台服务器。当然如果愿意的话，也完全可以用物理服务器来实现，不过这样做的话，在增加了扩展性的同时也增加了 Fabric 的操作难度。

以上描述的都是基于 FC 交换架构的网络，即 Fabric(FC 交换网络)。对于 FC 仲裁环架构的网络没有名称注册过程，环上的每个节点都对环上其他节点了如指掌，可以对任何节点发起通信。



有些机制可以把环路和交换结构融合起来，比如形成 Private loop、Public loop 等，这方面会在下文中介绍。

FC 的链路层和网络层被合并成一层，统称 FC2。

8.1.4 传输层

FC 的传输层同样也与 TCP 类似，也对上层的数据流进行 Segment，而且还要区分上层程序，TCP 是利用端口号来区分，FC 则是利用 Exchange ID 来区分。每个 Exchange(上层程序)发过来的数据包，被 FC 传输层分割成 Information Unit，也就相当于 TCP 分割成的 Segment。然后 FC 传输层将这些 Unit 提交给 FC 的下层进行传输。下层将每个 segment 当作一个 Sequence，并给予一个 Sequence ID，然后将这个 Sequence 再次分割成 FC 所适应的帧，给每个帧赋予一个 Sequence Count，这样便可以保证帧的排列顺序。接收方接受到帧之后，会组合成 Sequence，然后根据 Sequence ID 来顺序提交给上层协议处理。图 8.10 显示了这种层次结构。图 8.11 为 FC 网络上的数据帧传输示意图。

传输层还有一个重要角色，就是适配上层协议，比如 IP 可以通过 FC 进行传输，SCSI 指令可以通过 FC 来传输等。FC 会提供适配上层协议的接口，就是 IP over FC 及 CSI over FC。这里，FC 只是给 IP 和 SCSI 提供了一种通路，一种传输手段，就像 IP over Ethernet 和 IP over ATM 一样。

FC 也是通过发送 ACK 帧来向对方发送确认信息的，这个和 TCP 实现思想一样。只不过一个 ACK 帧是 24 字节加上 CRC、SOF、EOF，一共 36 字节，而 TCP 的 ACK 帧为 14+20+20=54

字节。两者差别已经很明显了，两个帧看不出来，但是发送多了，差别就看出来了。要看累积效应。当然这么算是很粗略的，还需要包括进链路控制，帧间隙开销等。

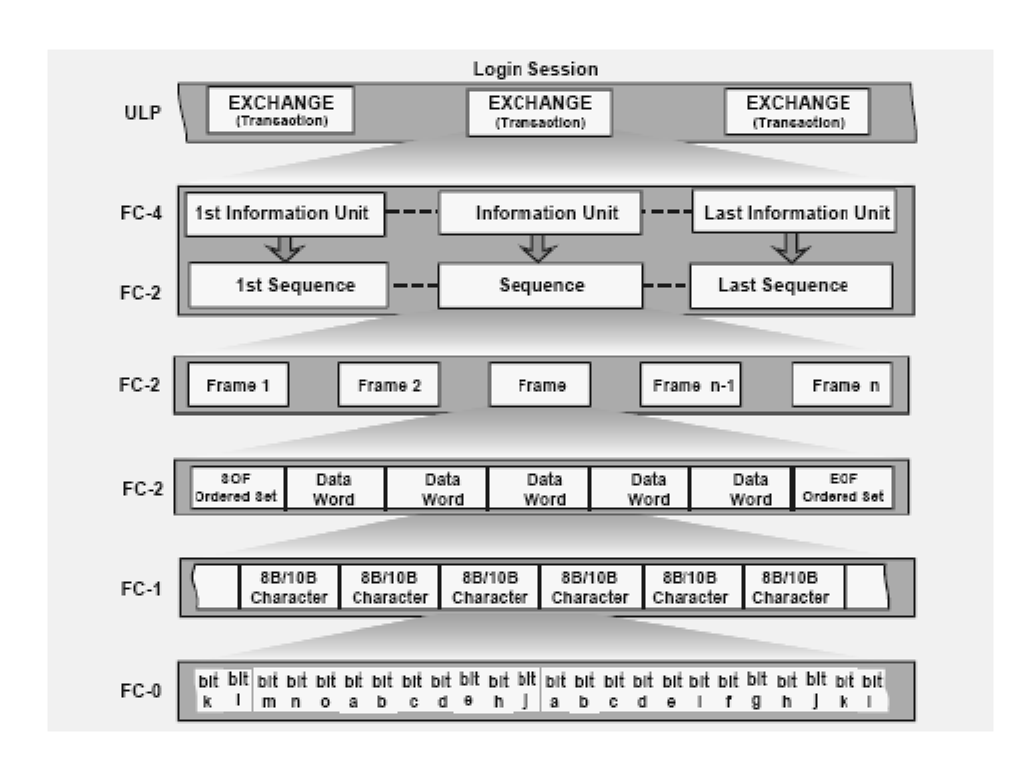


图8.10 FC 协议的层次结构

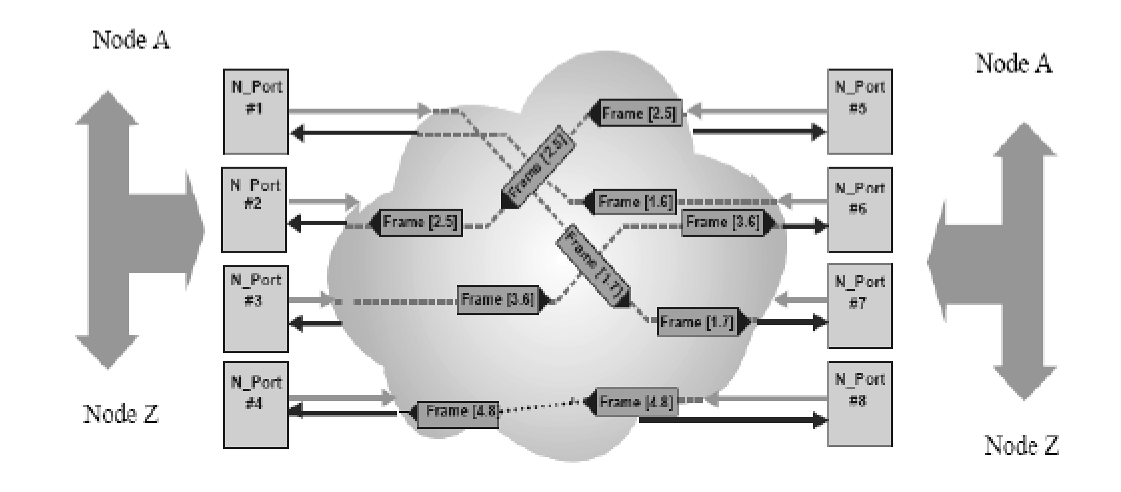


图8.11 Fabric 网络上的帧

在传输层上，FC 定义了几种服务类型，也就是类似 TCP/IP 协议中规定的 TCP、UDP。FC 协议中的 Class 1 服务类型是一种面向连接的服务，即类似电路交换的模式，为通信的双方保留一条虚电路，以进行可靠的传输。Class 2 类型提供的是一种带端到端确认的保障传输的服务，也就是类似 TCP。Class 3 类型不提供确认，类似 UDP。Class 4 类型是一个在一条连接上保留一定的带宽资源给上层应用，而不是像 Class 1 类型那样保留整个连接，类似 RSVP 服务。使用什么服务类型，会在端口之间进行 PLogin 的时候协商确定。

FC 传输层被定义为 FC4。

8.1.5 上三层

FC 协议的上三层表现为各种 Login 过程、包括名称服务等在内的各种服务等，这些都是与网络传输无关的，但是的确属于 FC 协议体系之内的，所以这些内容都属于 FC 协议的上三层。

8.1.6 小结

综上所述，FC 是一个高速高效，配置简单，不需要太多人为介入的网络。基于这个原则，为了进一步提高 FC 网络的速度和效率，在 FC 终端设备上，FC 协议的大部分逻辑被直接做到一块独立的硬件卡片当中，而不是运行在操作系统中。如果将部分协议逻辑置于主机上运行，会占用主机 CPU 内存资源。

TCP/IP 就是一种运行于主机操作系统上的网络协议，其 IP 和 TCP 或者 UDP 模块是运行在操作系统上的，只有以太网逻辑是运行在以太网卡芯片中的，CPU 从以太网卡接受到的数据是携带有 IP 头部及 TCP/UDP 头部的，需要运行在 CPU 中的 TCP/IP 协议代码来进一步处理这些头部，才能生成最终的应用程序需要的数据。

而 FC 协议的物理层到传输层的逻辑，大部分运行在 FC 适配卡的芯片中，只有小部分关于上层 API 的逻辑运行于操作系统 FC 卡驱动程序中，这样就使 FC 协议的速度和效率都较 TCP/IP 协议高。这么做，成本无疑会增加，但是这网络本来就不是为大众设计的，增加成本来提高速度和效率也是值得的。

8.2 FC 协议中的七种端口类型

在 FC 网络中，存在七种类型的接口，其中 N、L 和 N L 端口被用于终端节点，F、FL、E 和 G 端口在交换机中实现。

8.2.1 N 端口和 F 端口

N 端口和 F 端口专用于 Fabric 交换架构中。连入 FC 交换机的终端节点的端口为 N 端口，对应的交换机上的端口为 F 端口。N 代表 Node，F 代表 Fabric。用 N 端口模式连入 F 端口之后，网络中的 N 节点之间就可以互相进行点对点通信了。图 8.12 所示的是 N 端口和 F 端口的示意图。



图8.12 N 端口和 F 端口

8.2.2 L 端口

L 端口指仲裁环上各个节点的端口类型(LOOP)。环路上的所有设备可以通过一个 FCAL 的集线器相连，以使得布线方便，故障排除容易。当然，也可以使用最原始的方法，就是首尾相接。图 8.13 所示的是利用集线器连接的拓扑。

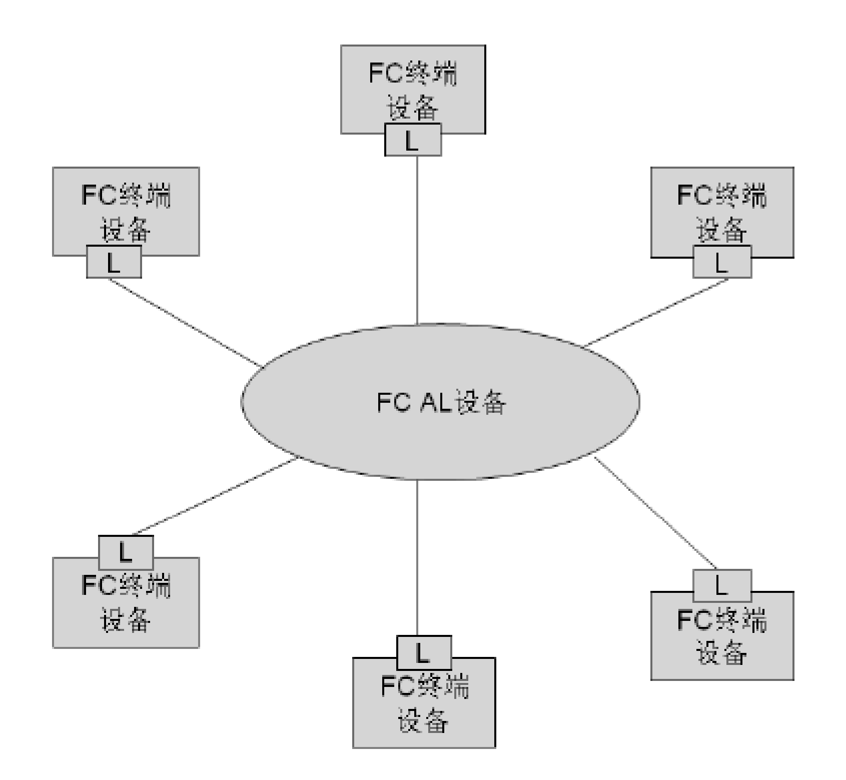


图8.13 基于 FCAL 集线器的 FCAL 环路连接

1. 私有环

私有环，就是说这个 FC 仲裁环是封闭的，只能在这个环中所包含的节点之间相互通信，而不能和环外的任何节点通信。

2. 开放环

这个环是开放的，环内节点不但可以和环内的节点通信，而且也可以和环外的节点通信。也就是说可以把这个环作为一个单元连接到 FC 交换机上，从而使得环内的节点可以和

位于 FC 交换机上的其他 N 节点通信。如果将多个开放环连接到交换机，那么这几个开放环之间也可以相互通信。

要实现开放环架构，需要特殊的端口，即下面描述的 NL 和 FL 端口。

8.2.3 NL 端口和 FL 端口

N L 端口是开放环中的一类端口，它具有 N 端口和 L 端口的双重能力。换而言之， N L 端口支持交换式光纤网登录和环仲裁。而 FL 端口是 FC 交换机上用于连接开放仲裁环结构的中介端口。

开放环内可以同时存在 NL 节点和 L 节点，而只有 NL 节点才能和环外的、位于 FC 交换结构中的多个 N 节点或者其他类型节点通信。NL 节点也可以同时和 L 节点通信。图 8.14 为 NL 和 FL 端口示意图。

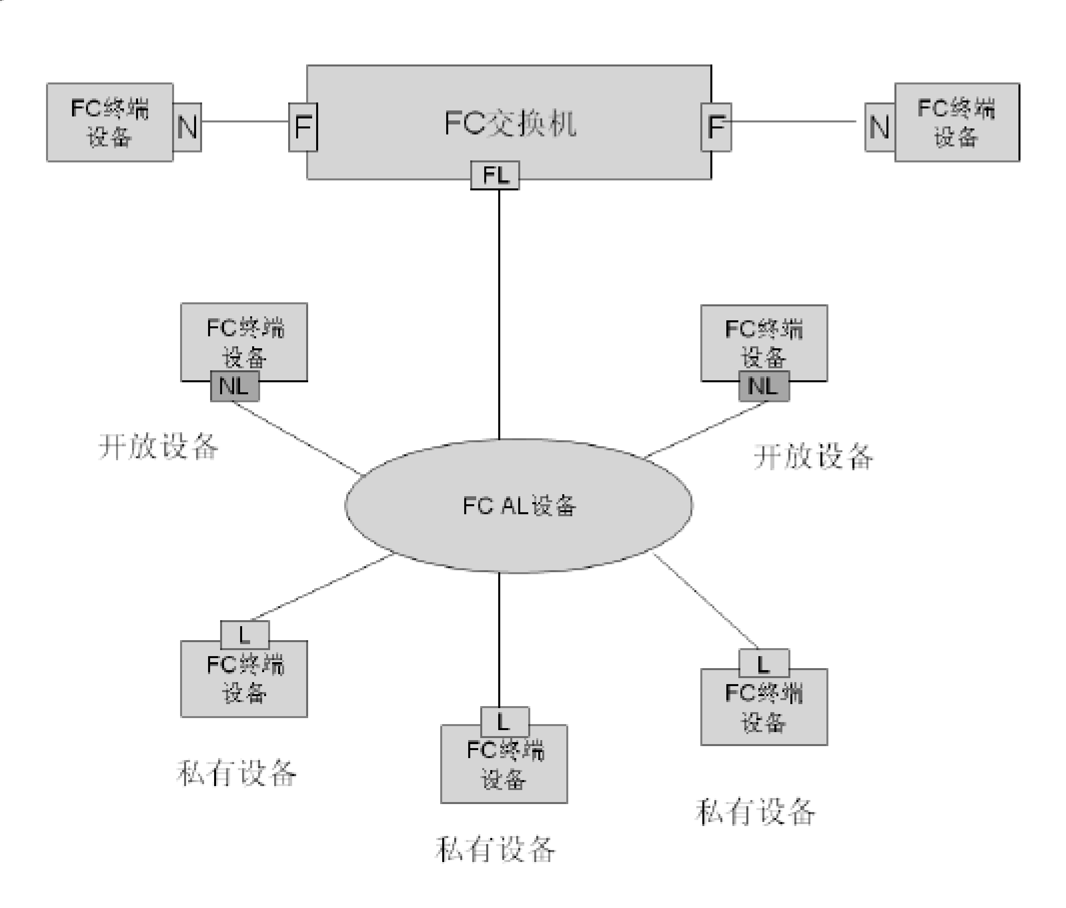


图8.14　NL 和 FL 端口示意图

开放环的融合机制

FC-SW 设备的工作方式是它会登录到网络(FLOGI)，并在 Name Server 中注册(PLOGI)。设备要传输数据时会先到 Name Server 查询 Target 的设备，然后到目标设备进行(PRLI)，最后传输数据。

FC-AL 的设备工作方式与此完全不同，在环路的初始化(LIP)过程中，生成一个环路上所有设备地址的列表，被称作 AL_PA，并存储在 Loop 中的每个设备上。当设备要与目标主机通信时，会到 AL_PA 中查询目标主机，然后根据地址进行通信。

要让一个私有环中的设备和 Fabric 中的设备达到相互通信，必须采用协议转换措施，因为 FC AL 和 FC Fabric 是两套不同逻辑体系。



在本书第 13 章论述了关于“协议之间相互作用”即“协议杂交”方面的内容。如果阅读过那一章，再回头来研究，我们可以发现，NL 端口和 FL 端口之间，完全就是一种 Tunnel 模式，他们利用 FC AL 的逻辑，承载 FC Fabric 的逻辑，也就是踩着 AL 走 Fabric。比如 Flogin、PLogin 等这些帧，都通过 AL 链路来发向 FL 端口，而整个环中其他节点，对这个动作丝毫不知道，也不必知道。

如果采用 MAP 方式达到两种协议形式的最大程度的融合，也是完全可以的。下面描述的这种模式，就是采用了 MAP 的思想。

这种 MAP 的模式使环内的任何 L 节点可以和环外的任何 N 节点之间就像对方和自己是同类一样通信。也就是说环内的 L 节点看待环外的 N 节点就像是一个不折不扣的 L 节点。反过来，环外的 N 节点看待环内的 L 节点就像是一个 N 节点一样。这个功能是通过在交换机上的 FL 端口实现的，这个端口承接私有环和 Fabric。在私有环一侧，它表现为 L 端口的所有逻辑行为，而对 Fabric 一侧，它则表现为 N 端口的行为，也就相当于一个 N-L 端口协议转换。这个接口可以把环外的 N 节点 “带”到私有环内，同时把环内的节点“带”到环外。环内的 L 节点根本不会知道它们所看到的其实是环外的 N 节点通过这个特殊的 L 端口仿真而来的。

当然也要涉及到寻址的 MAP，因为 Fabric 和 AL 的编址方式不同，所以需要维护一个地址映射，将环内的节点统统取一个环外的名字，也就是将 L 端口地址对应一个 N 端口地址，而这些地址都是虚拟的，不能和环外已经存在的 N 端口地址重合，这样才能让环外节点知道存在这么一些新加入 Fabric 的节点(其实是环内的 L 节点)。而要让环外节点知道这些新节点的存在，就要将这些新的节点注册到名称服务器上。因为 Fabric 架构中，每个节点都是通过查询名称服务器来获取当前 Fabric 中所存在的节点的。同样，要让环内的节点知道环外的 N 节点的存在，也必须给每个 N 节点取一个 AL 地址，让这些地址参与环的初始化，从而将这些地址加入到 AL 地址列表中。这样，环内的节点就能根据这个列表知道环内都有哪些节点了。

让各自都能看到对方，知道对方的存在，这只是完成了 MAP 的第一步。接下来，还要进行更加复杂的 MAP，即协议交互逻辑的 MAP。假如一个环内节点要和一个环外节点通信，这个环内节点会认为它所要通信的就是一个和它同类的 L 节点，所以它赢得环仲裁之后，会直接向这个虚拟 AL 地址发起通信。

这个虚拟 AL 地址对应的物理接口实际上是交换机上的仿真 L 端口，仿真 L 端口收到由环内节点发起的通信请求之后，便开始 MAP 动作。首先仿真 L 端口根据这个请求的目的地址，也就是那个虚拟地址，查找地址映射表，找到对应的 N 端口的 Fabric 地址。然后主动向这个 N 端口发起 PLogin 过程，也就是将 AL 的交互逻辑最终映射到了 Fabric 的交互逻辑。即 AL 向虚拟地址发起的通信请求，被仿真 L 端口 MAP 成了向真正的 N 端口发起 PLogin 请求，这就是协议交互逻辑的 MAP。请求成功之后，仿真 L 端口便一边收集环内 L 节点发来的数据，一边将数据按照 Fabric 的逻辑转发给真正的 N 端口。反之亦然，N 端口的逻辑，仿真 L 端口同样也会 MAP 成 AL 环的逻辑。这样，不管是环外的 N 端口还是环内的 L 端口，它们都认为它们正在和自己的同类通信。

图 8.15 所示为开放环与 Fabric 融合的示意图。

同样是将环接入 Fabric，开放环的扩展性就比私有环接入强。因为一个 NL 端口可以和环外的多个 N 端口通信。也就是说，NL 端口和 FL 端口可以看成是隐藏在环中的 N 端口和 F 端口。它们如果要通信，不能像直连的 N 和 F 端口那样直接进行 Fabric 登录，而必须先突破环的限制，即先要赢得环仲裁，再按照交换架构的逻辑进行 Fabric 登录，接着 N 端口登录，然后进程登录。而这一切，环内其他节点不会感知到。

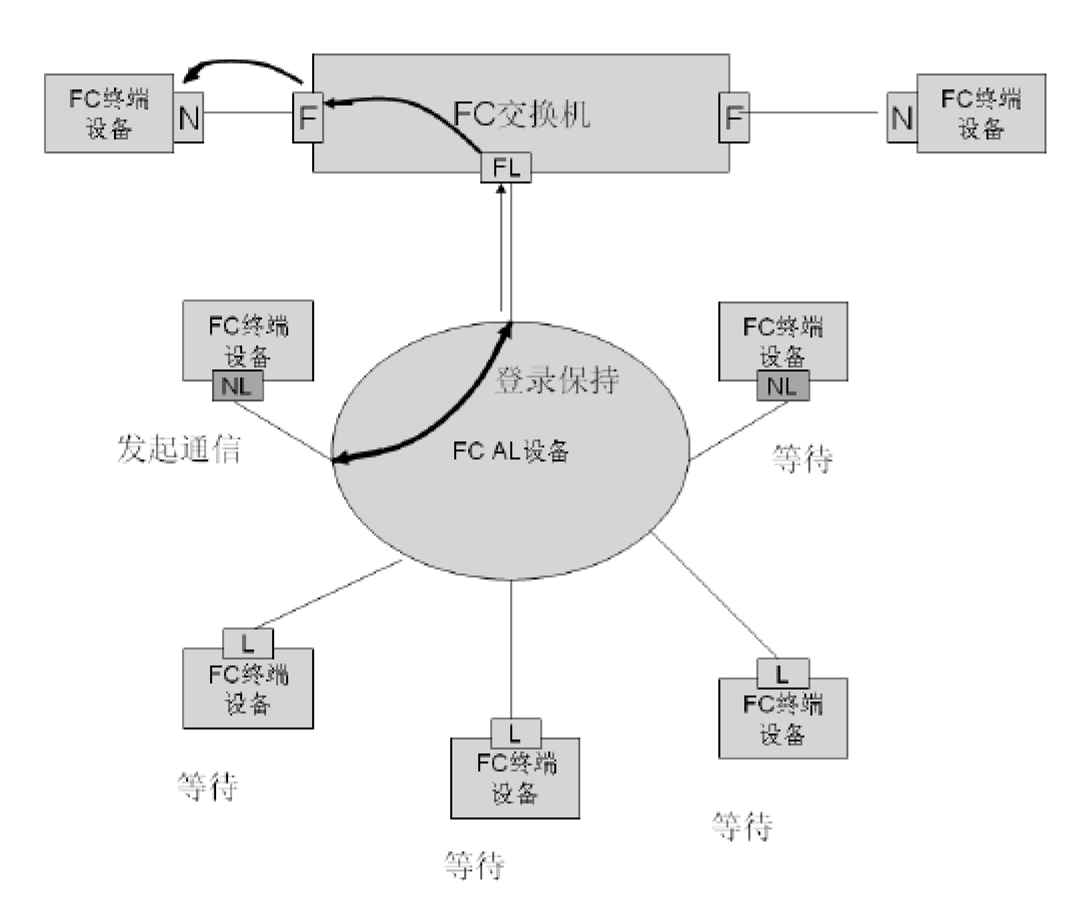


图8.15 开放环融合机制示意图

具有 NL 端口的设备既能和环内的 L 端口设备通信，又能和环外的 N 端口设备通信，同时具有 N 和 L 端口的逻辑，这一切都不需要仿真 MAP，只需要一个 Tunnel 过程即可。而环内的 L 节点如果想与环外的 N 端口通信，由于 L 节点自身没有 N 端口的逻辑，必须经过 FL 端口的 MAP 过程。所以，称具有 NL 端口的设备为 Public 设备，即开放设备。而称具有 L 端口的设备为 Private 设备，即不开放的私有设备。

8.2.4 E 端口

E 端口是专门用于连接交换机和交换机的端口。因为交换机之间级联，需要在级联线路上承载一些控制信息，比如选举协议、路由协议等。

8.2.5 G 端口

G 端口比较特殊，它是“万能”端口，它可以转变为上面讲到过的任何一种端口类型，按照所连接对方的端口类型进行自动协商变成任何一种端口。

终端节点端口编址规则

各种终端节点端口(N、NL、L)的 FC ID 地址都是 24 位(3 字节)长。但是 N 端口只使用 3 字节中的高 2 字节，即高 16 位；L 端口只使用 3 字节中的低 1 字节，即低 8 位；NL 端口使用全部 3 字节。没有被使用的字节值为 0。

产生这种编址机制不同的原因，是 3 种端口的作用方式不同。L 端口只在私有环内通信，而一个环的节点容量是 128 个，所以只用 8 位就可以表示了。N 端口由于处于 Fabric 交换架构中，节点容量很大，所以用了 16 位表示，最大到 65536 个节点。而 NL 端口，因为既处于环中，又要和 Fabric 交换架构中的节点通信，所以它既使用 N 端口的编址，又使用 L 端口的编址，所以用了全部 3 个字节。图 8.16 为端口编址示意图。

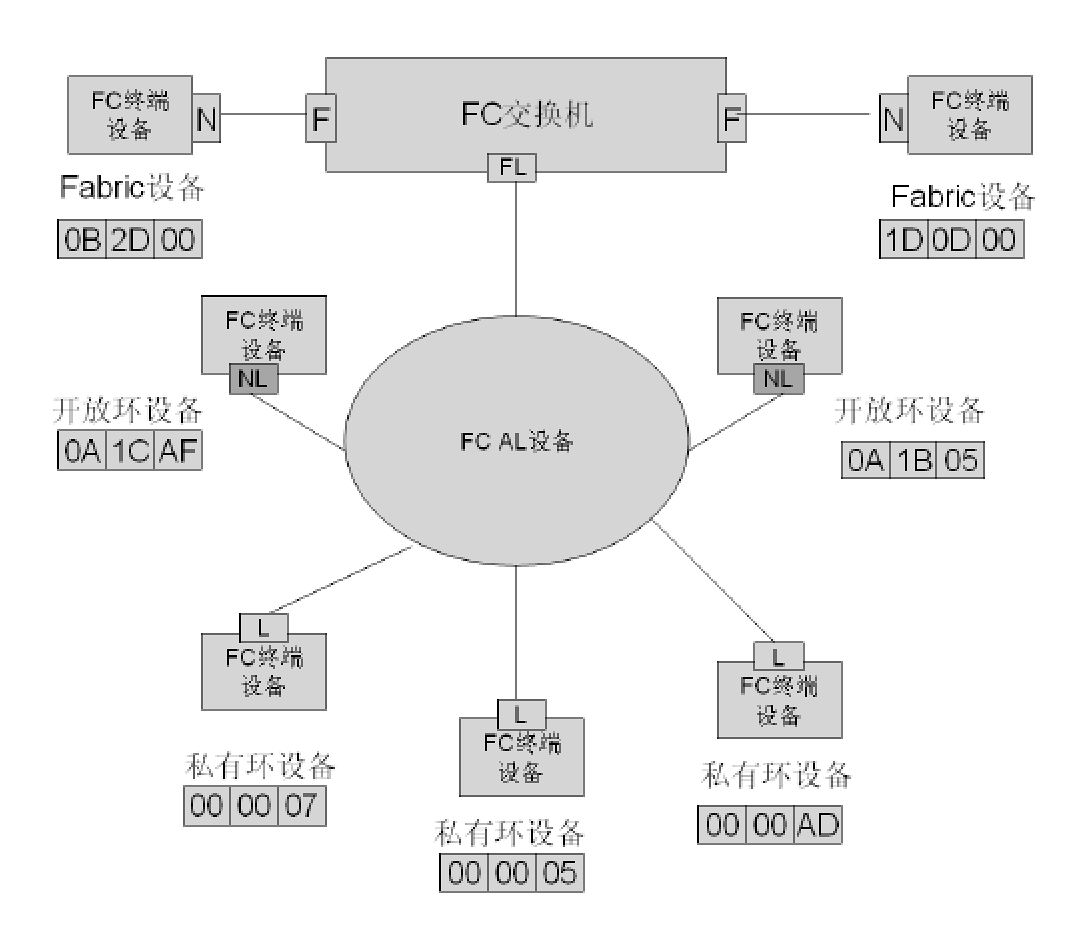


图8.16　三种 FC 节点类型的编址异同

任何设备都可以接入 FC 网络从而与网络上的其他 FC 设备通信，网络中的设备可以是服务器、PC、磁盘阵列、磁带库等。然而，就像以太网要求设备上必须有以太网接口才能连入以太网络一样，设备上必须有 FC 接口才可以连入 FC 网络。

8.3　FC 适配器

想进入 FC 网络，没有眼睛和耳朵怎么行呢。FC 网络的眼睛就是 FC 适配器，或者叫做 FC 主机总线适配器，即 FC HBA(Host Bus Adapter)。值得说明的是，HBA 是一个通用词，它不仅仅指代 FC 适配器，它可以指代任何一种设备，只要这个设备的作用是将一个外部功能接入主机总线。所以，PC 上用的 PCI/PCIE 网卡、显卡、声卡和 AGP 显卡等都可以叫做 HBA。

图 8.17 所示的就是 PCI 接口的 FC 适配器。

图 8.18 所示的是可以用来接入 FC 网络的各种线缆，可以看到 SC 光纤、DB9 铜线和 RJ45/47 线缆，它们都可以用于接入 FC 网络，只要对端设备也具有同样的接口。所以，千万不要认为 FC 就是光纤，这是非常滑稽的。

同样，也不要认为 FC 交换机就是插光纤的以太网交换机，这是个低级错误。称呼 FC 为光纤的习惯误导了不少人。FC 协议是一套完全独立的网络协议，比以太网要复杂得多。FC 其实是 Fibre Channel 的意思，由于 Fibre 和 Fiber 相似，再加上 FC 协议普遍都用光纤作为传输线缆而不用铜线，所以人们下意识的称 FC 为光纤通道协议而不是网状通道协议。但是要理解，FC 其实是一套网络协议的称呼，FC 协议和光纤或者铜线实际上没有必然联系。如果可能的话，也可以用无线、微波、红外线或紫外线等来实现 FC 协议的物理层。同样以太网协议与是否用光纤或者铜线、双绞线来传输也没有必然联系。

所以“FC 交换机就是插光纤的以太网交换机”的说法是错误的。同样“以太网就是双绞线”和“以太网就是水晶头”这些说法都是滑稽的。

FC 适配器本身也是一个小计算机系统，有自己的 CPU 和 RAM 以及 ROM。ROM 中存放 Firmware，加电之后由其上的 CPU 载入运行。可以说就是一个嵌入式设备，与 RAID 卡类似，只不过不像 RAID 卡一样需要那么多的 RAM 来做为数据缓存。

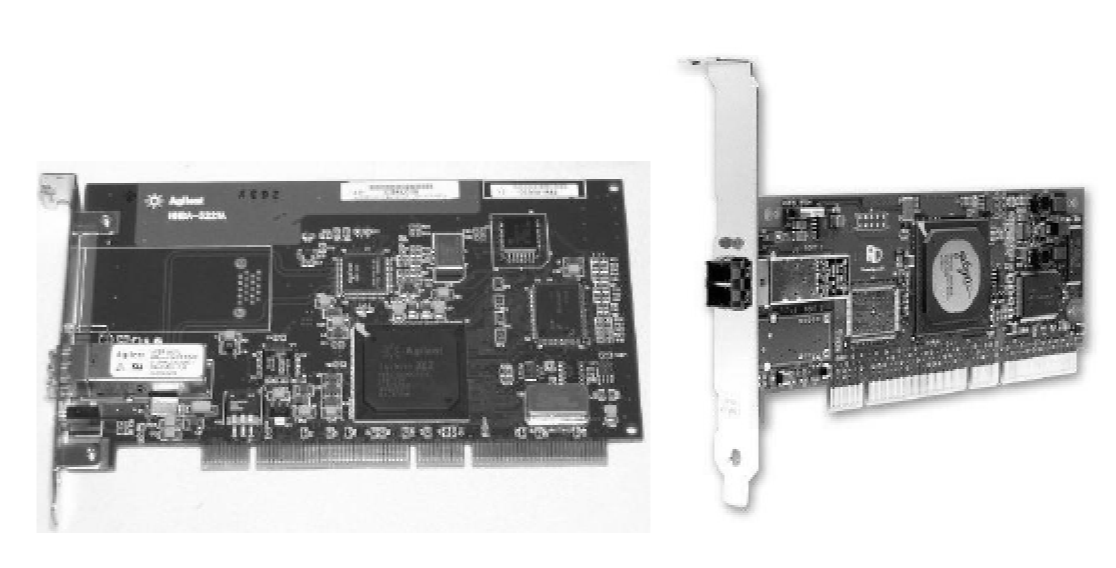




图8.17 FC 适配卡

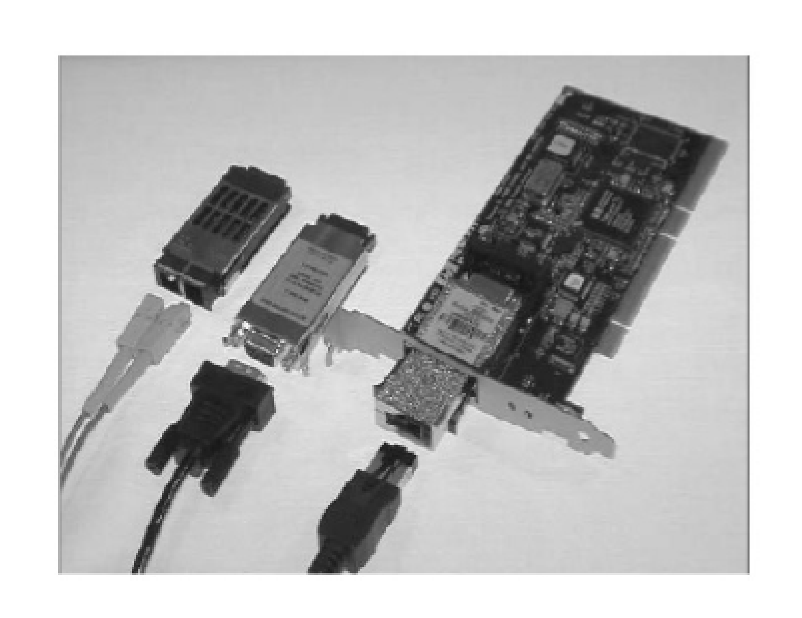


图8.18 各种接口的 FC HBA

8.4 改造盘阵前端通路——SCSI 迁移到 FC

现在是考虑把原来基于并行 SCSI 总线的存储网络架构全面迁移到 FC 提供的这个新的网络架构的时候了！

但是 FC 协议只是定义了一套完整的网络传输体系，并没有定义诸如 SCSI 指令集这样可用于向磁盘存取数据的通用语言。而目前已经有了两种语言，一种是 ATA 语言(ATA 指令集)，另一种就是 SCSI 语言(SCSI 指令集)。那么 FC 是否有必要再开发第三种语言？完全没有必要了。SCSI 指令集无疑是一个高效的语言，FC 只需要将 SCSI 语言拿来用就可以，但必须将这种语言承载于新的 FC 传输载体进行传送。

SCSI 协议集是一套完整而不可分的协议体系，同样有 OSI 中的各个层次。物理层使用并行传输。SCSI 协议集的应用层其实就是 SCSI 协议指令，这些指令带有强烈应用层语义。而我们要解决的就是如何将这些指令帧传送到对方。早期并行 SCSI 时代，就是用 SCSI 并行总线技术来传送指令，这个无疑是一个致命的限制。随着技术的发展，并行 SCSI 总线在速度和效率上已经远远无法满足要求。好在 SCSI-3 协议规范中，将 SCSI 指令语义部分(OSI 上三层)和 SCSI 底层传输部分(OSI 下四层)分割开了，使得 SCSI 指令集可以使用其他网络传输方式进行传输，而不仅仅限于并行 SCSI 总线了。

FC 的出现就是为了取代 SCSI 协议集的底层传输模块，由 FC 协议的底层模块担当传输通道和手段，将 SCSI 协议集的上层内容传送到对方。可以说是 SCSI 协议集租用了 FC 协议，将自己的底层传输流程外包给了 FC 协议来做。

FC 协议定义了在 FC4 层上的针对 SCSI 指令集的特定接口，称为 FCP，也就是 SCSI over FC。由于是一个全新的尝试，所以 FC 协议决定先将连接主机和磁盘阵列的通路，从并行 SCSI 总线替换为串行传输的 FC 通路。而盘阵后端连接磁盘的接口，还是并行 SCSI 接口不变。

从图 8.19 中可以看到，连接主机的前端接口已经替换成了 FC 接口，原来连接在主机上的 SCSI 适配器也被替换成了 FC 适配器。

经过这样改造后的盘阵，单台盘阵所能接入磁盘的容量并没有提升，也就是说后端性能和容量并没有提升，所提升的只是前端性能。因为 FC 的高效、高速和传输距离，远非并行 SCSI 可比。

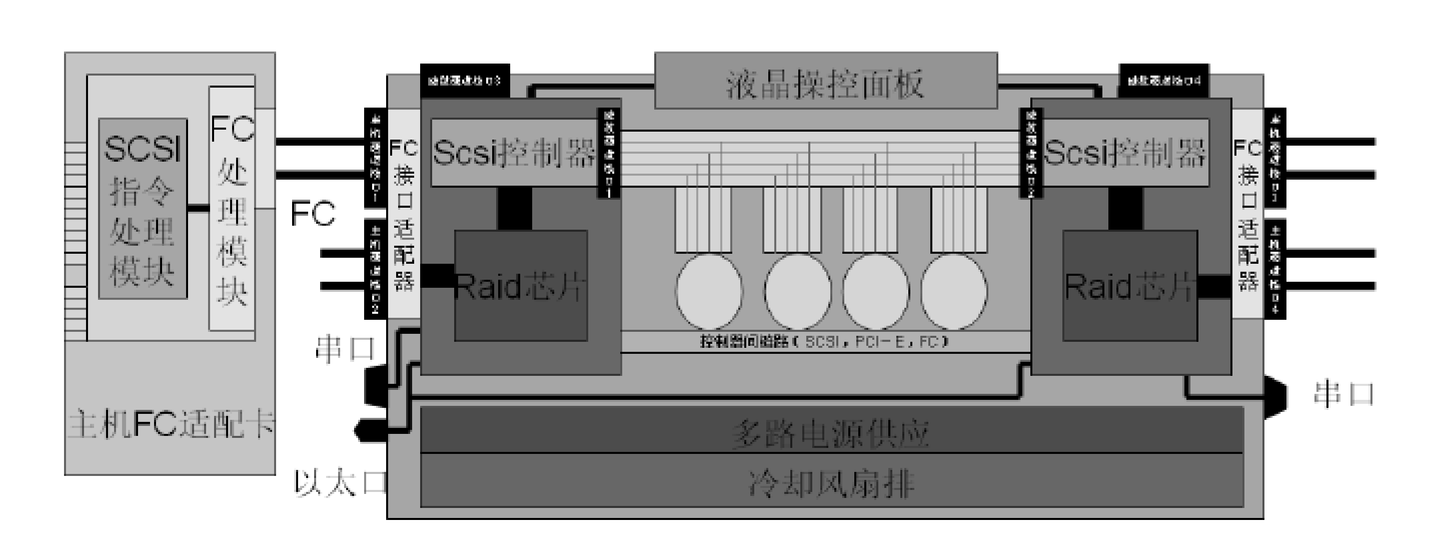


图8.19　前端 FC、后端 SCSI 架构的盘阵示意图

> 提示
> 理解：虽然链路被替换成了 FC，但是链路上所承载的应用层数据并没有变化，依然是 SCSI 指令集，和并行 SCSI 链路上承载的指令集一样，只不过换成 FC 协议及其底层链路和接口来传输这些指令以及数据而已。

从图 8.19 中可以看到，不管是主机上的 FC 适配器还是盘阵上的控制器，都没有抛弃 SCSI 指令集处理模块，被抛弃的只是 SCSI 并行总线传输模块。也就是抛弃了原来并行 SCSI 协议集位于 OSI 的下四层(用 FC 的下四层代替)，保留了整个 SCSI 协议的上三层，也就是 SCSI 指令部分。

将磁盘阵列前端接口用 FC 替代之后，极大地提高了传输性能以及传输距离，原来低效率、低速度和短距离的缺点被彻底克服了。

8.5　引入 FC 之后

1. 提高了扩展性

FC 使存储网络的可扩展性大大提高。如图 8.20 所示，一台盘阵如果只提供一个 FC 前端接口，同样可以连接多台主机，办法是把它们都连接到一台 FC 交换机上。就像一台机器如果只有一块以太网卡，而没有以太交换机或 HUB 的话，那么只能和一台机器相连。如果有了以太网交换机或 HUB，它就可以和 N 台机器连接。使用 FC 交换机的道理一样，这就是引入包交换网络化所带来的飞跃。

多台主机共享一台盘阵同时读写数据，这个功能在并行 SCSI 时代是想都不敢想的。虽然并行 SCSI 总线网络可以接入 16 个节点，比如 15 台主机和一台盘阵连入一条 SCSI 总线，这 15 台主机只能共享这条总线的带宽，假设带宽为 320MB/s，如果 15 台主机同时读写，则理论上平均每台主机最多只能得到 20MB/s 的带宽。而这只是理论值，实际加上各种开销和随机 IO 的影响，估计每台主机能获得的吞吐量会不足 10MB/s。再加上 SCSI 线缆最长不能超过 25 米，用一条宽线缆去连接十几台主机和盘阵的难度可想而知。

而引入 FC 包交换网络之后，首先是速度提升了一大截，其次由于其包交换的架构，可以很容易的实现多个节点向一个节点收发数据的目的。

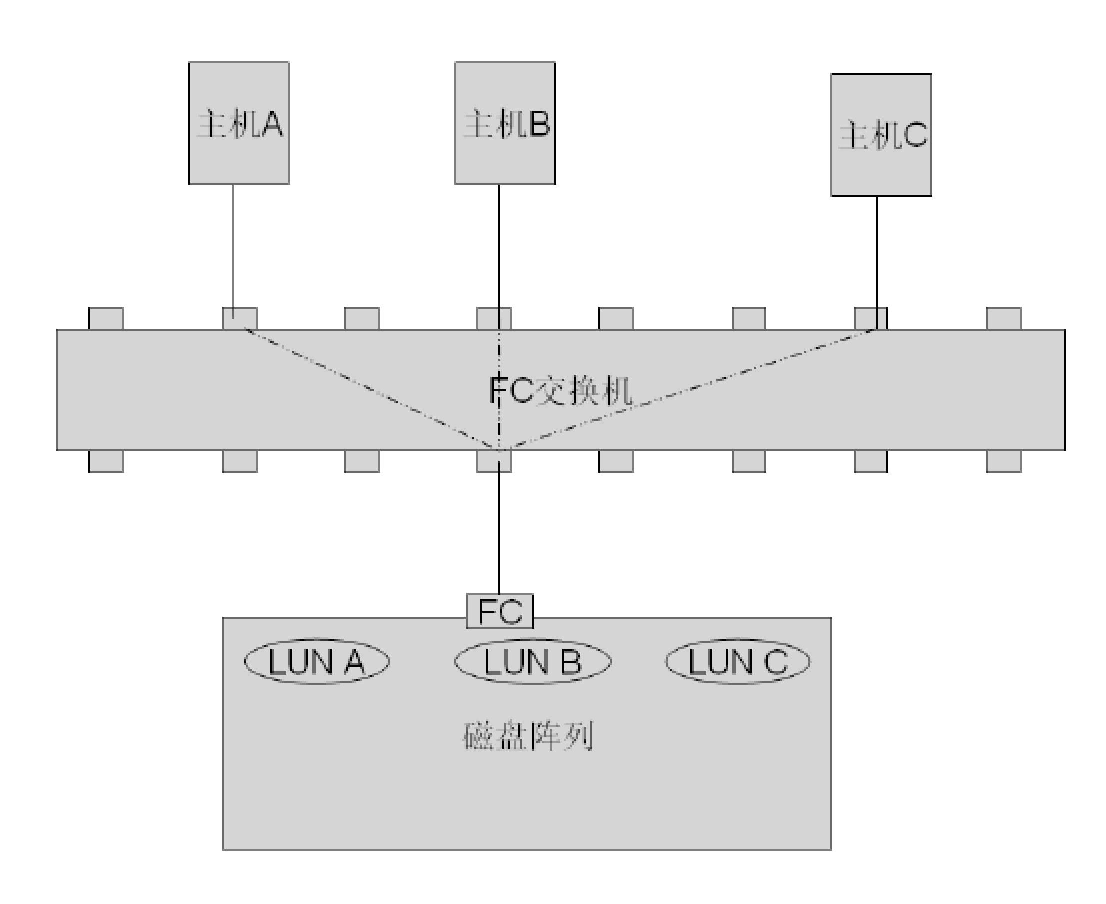


图8.20 多主机共享盘阵

2. 增加了传输距离

FC 携带有现代通信的特质，比如可以使用光纤。而这就可以使主机和与远隔几百米甚至上千米(使用单模光缆)之外的盘阵相连并读写数据。

3. 解决安全性问题

可能很多人还会有疑问，在图 8.20 所示的拓扑中，多个主机共用一台只有一个外部接口的盘阵不会冲突么？当然不会。第一，交换机允许多个端口访问同一个端口是一个分时复用的包交换过程，这个是无庸置疑的。第二，盘阵上的 FC 前端接口允许多个其他端口进行 Port Login 过程。那么盘阵上的逻辑磁盘 LUN 可以同时被多个主机访问么？完全可以。SCSI 指令集中有一个选项，即独占式访问或者共享式访问。

(1) 独占式访问，即只允许第一个访问某个目标节点的节点保持对这个目标节点的访问，第二个节点要向这个目标节点发起访问请求，每不被允许，除非上一个节点发出了释放指令。独占模式下，每台主机每次访问目标前都需要进行 SCSI Reserve，使用完后再进行 SCSI Release 释放 SCSI 目标，这样其他节点才能访问那个目标。

(2) 共享式访问，即允许任何人来访问，没有任何限制。

所以，盘阵上的任何 LUN 都可以被多台主机通过一个前端接口或者多个前端接口访问。这是一个优点，也是一个隐患。因为多个主机在没有相互协商和同步的情况下，一旦对同一个 LUN 都进行写操作的话，就会造成冲突。比如两台 Windows 主机正处于运行状态，它们都通过 FC 适配卡识别到了磁盘阵列上的同一个 LUN。此时主机 A 向这个 LUN 上写了一个文件，假设主机 B 已经将文件系统的元数据读入了内存，磁盘上的数据被主机 A 更改这个动作主机 B 是感受不到的。隔一段时间之后，主机 B 可能将文件系统缓存 Flush 到磁盘，此时可能会抹掉这个文件的元数据信息。

所以，在没有协商和同步机制的两台主机之间共享一个 LUN 是一件可怕的事情。要解决这个问题，可以每次只开一台机器，主机 B 想访问就必须把主机 A 关机或卸载该卷，然后主机 B 开机或挂载该卷，这样才能保证数据的一致性。但这样有点过于复杂。第二种办法就是使两台或者多台机器同时开机或同时挂载该卷，而让机器上的文件系统之间相互协商同步，配合运作。我写入的东西会让你知道。如果我正在写入，那么你不能读取，因为你可能读到过时的信息。

在文件系统上增加这种功能，需要对文件系统进行修改，或直接安装新的文件系统模块。这种新的文件系统叫做集群文件系统，能保证多个机器共享一个卷，不会产生破坏。

有些情况确实需要让两台机器同时可以访问同一个卷(如集群环境)，但是大多数情况下是不需要共享同一个卷的，每台机器拥有各自的卷，都只能访问属于自己的卷，这样不就太平了么？

是的，要做到这一点有两种方法。分析从主机到盘阵上的 LUN 的通路，可以发现通路上有两个部件，第一个部件是 FC 网络交换设备，第二个部件就是磁盘阵列控制器。可以在这两个部件上做某种“隐藏”或者“欺骗”，让主机只能对属于它自己的 LUN 进行访问。

(1) 在磁盘阵列控制器上做“手脚”。

SCSI 指令集中有一条指令叫做 Report LUN，也就是在 SCSI 发起端和目标端通信的时候，由发起端发出这条指令，目标端在接收到这条指令之后，就要向发起端报告自己的 LUN 信息。可以在这上面做些手脚，骗发起端一把。当发起端要求 Report LUN 的时候，盘阵控制器可以根据发起端的惟一身份(比如 WWPN 地址)，提供相应的 LUN 报告给它。

比如针对主机 A，控制器就报告给它 LUN1、LUN2、LUN3。虽然盘阵上还配置很多其他的卷，比如 LUN4、LUN5、LUN6 等，但是如果告诉控制器，让它根据一张表 8.1 所示的映射表来判断应该报告给某个主机哪个或哪些 LUN 的话，控制器就会乖乖的按照指示来报告相应的 LUN 给相应的主机。

表 8.1 LUN 映射表

针对哪个主机(WWPN 地址)	报告哪个或者哪些 LUN
主机 A WWPN 地址：00-16-E3-6E-78-05-0A-FD	LUN1、LUN2、LUN3
主机 B WWPN 地址：00-16-E3-6E-78-05-0A-0A	LUN4、LUN5
主机 C WWPN 地址：00-16-E3-6E-78-05-0A-3B	LUN6

如果某个主机强行访问不属于它的 LUN，盘阵控制器便会拒绝这个请求。上面那张映射表完全需要人为配置，因为盘阵控制器不会知道我们的具体需求。所以对于一个盘阵来说，要想实现对主机的 LUN 掩蔽，必须配置这张表。

盘阵上的这个功能叫做 LUN Masking(LUN 掩蔽)，也就是对特定的主机报告特定的 LUN。这样可以避免“越界”行为，也是让多台主机共享一个盘阵的方法，从而让多台主机和平共享一台盘阵资源。毕竟，对于容量动辄几 TB 甚至几百 TB 的大型盘阵来说，如果不加区分的让所有连接到这台盘阵的主机都可以访问到所有的卷是没有必要的，也是不安全的。

不仅 FC 接口盘阵有这个功能，SCSI 前端接口盘阵照样可以实现这个功能，因为这是 SCSI 指令集的功能，而不是传输总线的功能。不管用什么来传输 SCSI 指令集，只要上面能承载 SCSI 指令集，那么指令集中所有功能都可用。

磁盘阵列除了可以将某些 LUN 分配给某个主机之外，还可以配置选择性的将某个或某些 LUN 分配到某个前端端口上。也就是说，设置前端主机只有从某个盘阵端口进入才能访

问到对应的 LUN，从盘阵前端其他端口访问不到这些 LUN。有些双控制器的盘阵可以定制策略将某些 LUN 分配到某个控制器的某些端口上。LUN Masking 的策略非常灵活，只要有需求就没有开发不出来的功能。

总之，可以把 LUN 当作蛋糕，有很多食客(主机)想吃这些蛋糕。然而，食客要吃到蛋糕，需要首先通过迷宫(FC 网络)，然后到达一个城堡(磁盘阵列)。城堡有好几个门(盘阵的前端接口)，如果城堡的主人很宽松，会把所有蛋糕分配到所有门中，从任何一个门进入都可以吃到所有蛋糕。如果主人决定严格一些，那么他也许会将一部分蛋糕分配到 1 号门，另一部分蛋糕分配到 2 号门。如果主人非常非常严格，那他会调查每个食客的身份，然后制定一个表格，根据不同身份来给食客不同的蛋糕。

(2) 在 FC 交换设备上做“手脚”。

我们前面提到过 ZONE。ZONE 就是在 FC 网络交换设备上阻断两个节点间的通路，这样某些节点就根本无法获取并访问到被阻断的其他节点，也就识别不到其上的 LUN 了。LUN masking 只是不让看见某个节点上的某些 LUN 而已，而 ZONE 的做法更彻底，力度更大。

ZONE 有软 ZONE 和硬 ZONE 之分。软 ZONE 就是在名称服务器上做手脚，欺骗进行名称注册的节点，根据 ZONE 配置的信息向登录节点通告网络上的其他节点以及资源的信息。硬 ZONE 就是直接把交换机上某些端口归为一个 ZONE，另一些端口归为另一个 ZONE，在两个 ZONE 之间完全底层隔离，端口之间都不能通信，如图 8.21 所示。

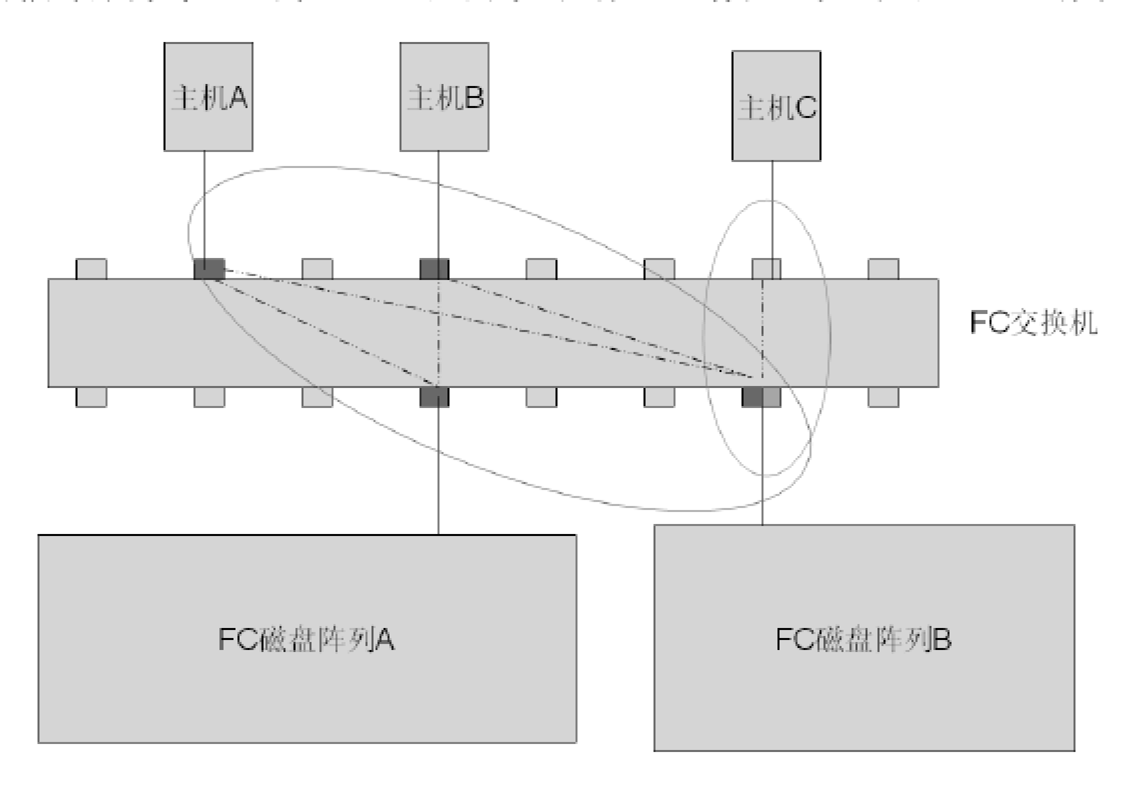


图8.21 ZONE 示意图

图中有两个 ZONE，FC 盘阵 B 所连接的交换机端口既在左边的 ZONE 里，又在右边的 ZONE 中，这样是允许的。这个例子中，主机 C 是无法和盘阵 A 通信的，它只能识别到盘阵 B 上的 LUN。

有了 LUN Masking 和 ZONE，FC 网络的安全就得到了极大的保障，各个节点之间可以按照事先配置好的规则通信。

4. 多路径访问目标

再来看一下图 8.22。这是一个具有双控制器的盘阵，两个控制器都连接到了交换机上，而且每个主机上都有两块 FC 适配卡，也都连接到了交换机上。前文说过，如果在盘阵上没有做 LUN Masking 的策略，而在 FC 交换机上也没有做任何 ZONE 的策略，则任何节点都可以获取到网络上所有其他节点的信息。

假设盘阵上有一个 LUN1 被分配给控制器 A，LUN2 被分配给控制器 B，那么可以计算

出来，每个主机将识别到 4 块磁盘。因为每个主机有两块 FC 适配卡，每个适配卡又可以识别到控制器 A 上的 LUN1 和控制器 B 上的 LUN2。也就是说，每台主机会识别到双份冗余的磁盘，而主机操作系统对这一切一无所知，它会认为识别到的每块磁盘都是物理上独立的，这样很容易造成混乱。

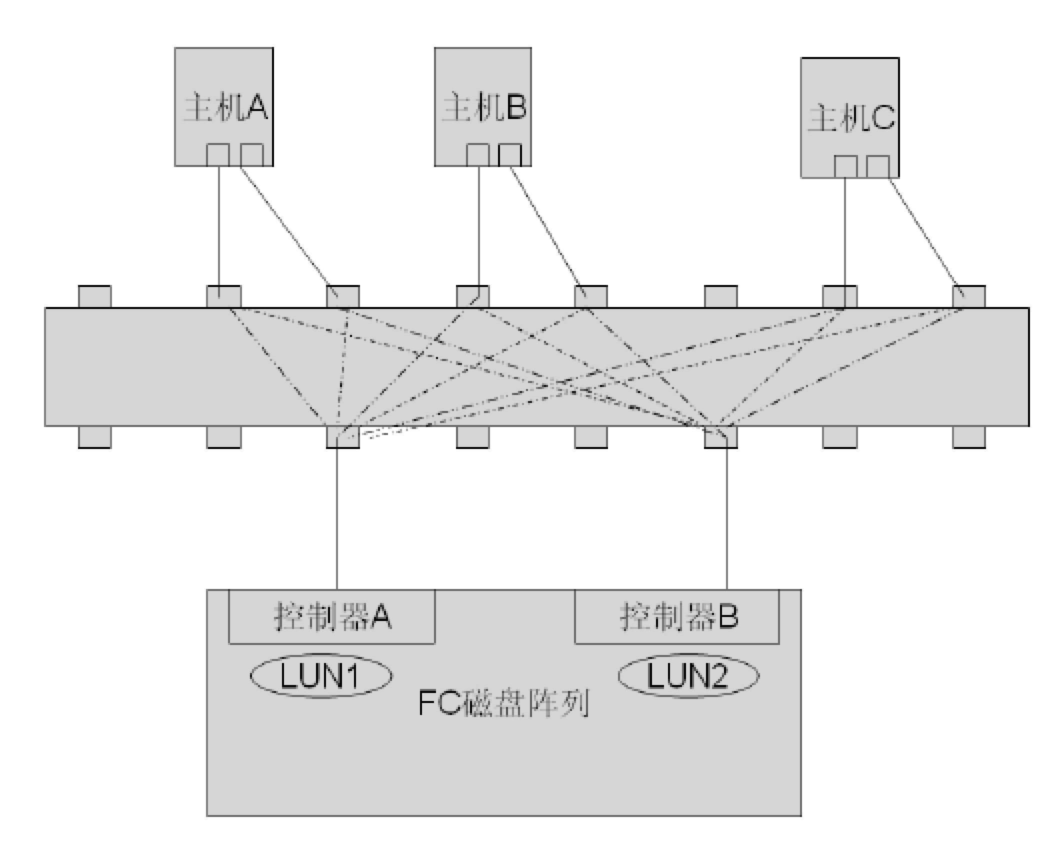


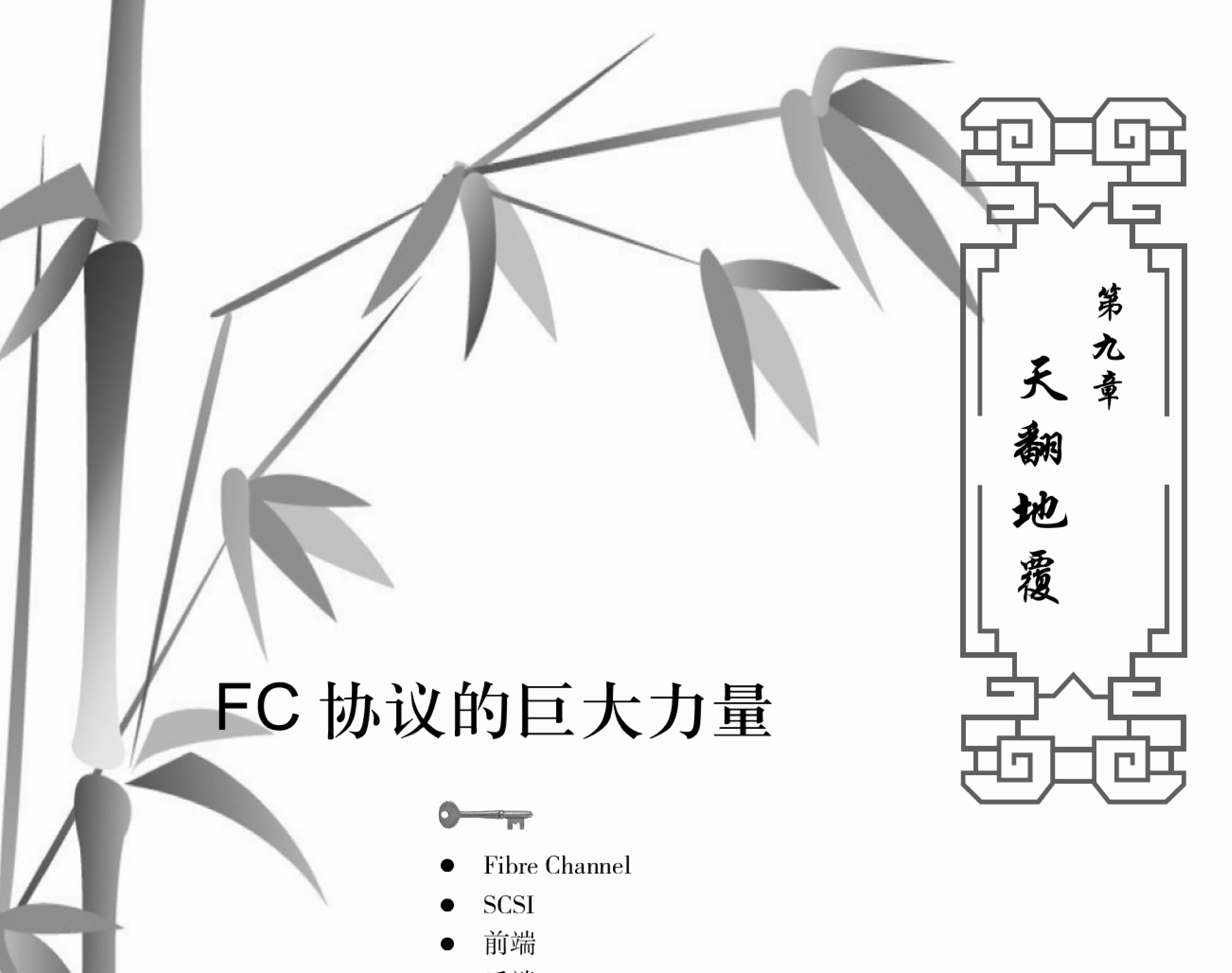
图8.22　多路径访问示意图

既然会造成混乱，那么为何要在一台主机上安装两块 FC 适配卡呢？就是为了冗余，以防止单点故障。一旦某块 FC 卡出现了故障，另一块卡依然可以维持主机到盘阵的通路，数据流可以立即转向另外一块卡。

如何解决操作系统识别出多份磁盘这个问题呢？办法就是在操作系统上安装软件，这个软件识别并分析 FC 卡提交上来的 LUN。如果是两个物理上相同的 LUN，软件就向操作系统卷管理程序提交单份 LUN。如果某块 FC 卡故障，只要主机上还有其他的 FC 卡可以维持到 FC 网络的通路，那么这个软件依然会向操作系统提交单份 LUN。一旦所有 FC 卡全都故障了，主机就彻底从 FC 网络断开了，这个软件也就无法提交 LUN 了，操作系统当然也识别不到盘阵上的 LUN 了。

此外，如果盘阵的某个控制器接口发生故障，主机同样可以通过这个软件立即重定向到另一个备份控制器，使用备份控制器继续访问盘阵。

这种软件叫做“多路径”软件，每个盘阵开发商都会提供自己适合不同操作系统的多路径软件。多路径软件除了可以做到冗余高可用性的作用之外，还可以做到负载均衡。因为主机上如果安装了多块 FC 适配卡，数据就可以通过其中任何一块卡到达目的，这样就分担了流量。



第九章 天翻地覆

FC 协议的巨大力量

- Fibre Channel
- SCSI
- 前端
- 后端
- 机头
- 扩展柜
- FC 磁盘

话说 FC 协议横空出世，在江湖上引起了轩然大波，各门派纷纷邀请 FC 协议来参与存储磁盘阵列的制造，用 FC 协议实现盘阵与主机的连接。

以前并行 SCSI 的时代已经结束，那又宽又短的电缆终于可以彻底将其抛弃，取而代之的是细长的光纤。

9.1 FC 交换网络替代并行 SCSI 总线的必然性

历史是不断前进的，事物也是不断发展的，新技术必定取代旧技术。FC 取代并行 SCSI 总线有两个根本原因。

9.1.1 面向连接与面向无连接

在并行 SCSI 总线时代并没有复杂的链路层协议，“连”就体现在线缆上，就像连接 CPU 和北桥之间的铜线一样，只不过 SCSI 线缆被做成了柔软的外置线缆而已。基本在这种短线缆上可以不必考虑通信层面的内容，因为距离很短，线路是稳定的，不需要加入诸如传输保障机制之类的东西。同时，这种情况也相当于面向连接的电路交换，通信的双方要预先建立一条物理上的通路(虚电路)，不管有没有数据流，这条通路总是存在，且带宽固定，别人也抢不走这条电路的使用权，这就给通信双方提供了最大的质量和稳定性保证。在这样的链路上，不需要过多的底层传输协议开销。

相反，在面向无连接的包交换网络中，数据流被封装成数据包，传输保障和流量控制等因素就显得十分重要了。因为此时网络是共享的，网络按照 Best Effect 尽力而为的转发数据包。以太网和 FC 交换网络都属于这种面向无连接的技术(FC 中 Class1 类型服务除外)。

而电话交换网、并行 SCSI 总线网就属于面向连接的网络。当你提机拨号的时候，电话局的电话交换机便会在你和你通信的对方之间建立起一条物理电路，从而使双方通信。



大家可以观察一下电话交换机，每当有外线拨入的时候，就可以听到交换机里有吧嗒吧嗒的声音，这就是交换机在做继电器开合动作。

对于并行 SCSI 总线，当通信发起者需要和某个节点通信的时候，它会申请总线仲裁，在获得总线资源之后，便直接和对方发起通信，此时并没有一个显式的连接建立的过程。物理通路总是存在的，在任意两点之间都存在，只不过这个通路是个总线，是大家共享的，需要通过仲裁来获得总线使用权，也就等价于建立独享连接。SCSI 指令和数据可以直接在这个总线上传递，并不需要过多的额外的协议开销。

1. 面向连接的致命弱点

但是面向连接的通信有三个致命弱点。

面向连接网络的第一个弱点，就是资源浪费。特别是在交换环境中，由于不管路径上有没有数据传输，这条预先建立的连接必须保持并且只给特定的通信双方使用，其他节点的通信不管数据多么拥塞都不能使用这条路径。面向连接的网络好比一个城市的公交系统，每条公交线路都是固定的，不管这条线路上的客流量多少，就算没有人坐这条线路了，公交车也要来回跑。而面向无连接的网络就好比出租车。在没有人的时候，出租车可以等待客人到来。一旦有客人到来，出租车便会上路，而且路线不是固定的，司机可以按照目前道路流量情况，选择空闲的道路前往目的地。

面向连接的网络的第二个弱点是维持和维护这条连接所耗费的成本高。通信双方距离近，没什么问题。但是一旦距离很远，要维持这条物理连接，就需要很高的成本。要解决长距离传输的干扰问题、需要中继等，这也是长途电话费居高不下的一个原因。

面向连接网络的第三个缺点，就是缺乏高可用性。一旦建立好的虚电路因为某种原因断开了，就需要通信发起者重新建立电路才能继续双方的通信。这种现象在打长途电话的时候经常会遇到，此时不得不重新拨打。而对于包交换网络，通信双方没有一条固定的数据流路径，交换或路由设备会自行判断数据流应当通过哪条路径到达对方。一旦某条动态的路径不再可用，交换设备会立即选用其他可以到达对方的路径，而这个短时的中断所造成的影响会交给通信双方运行的传输保障协议来处理，丢失的数据包会被重传。而用户对此不必关心，最多会感觉有短暂延迟，而不必重新和对方建立连接。

2. 面向无连接的优势

面向无连接的包交换网络比面向连接的网络有很多优势。面向无连接的包交换网络是网络通信的一种趋势，目前的 VOIP、IPTV 等应用都是想利用包交换网络来代替普通的电话交换网络和有线电视网络。

不要把“面向无连接”和“TCP 是有连接的”混淆在一起。TCP 是一个端到端的协议，它运行于通信双方，而不是通信所经过的网络设备上。TCP 的连接不是物理连接而是逻辑连接。TCP 其实就是一个状态，本身保持一个状态机用来侦测双方的数据流是否成功发送或者接收。实际通信两点间的连接可以经过包交换网络，同样也可以经过面向连接的网络。也就是说，“面向连接”和“面向无连接”是指链路层的概念，而 TCP 是传输层的概念。

9.1.2 串行和并行

串行传输在长距离高速传输方面，也必将取代并行传输。

并行 SCSI 总线就是一种面向连接的并行的共享总线技术。其趋势就是必将被高速串行的、面向无连接的网络通信技术所取代。取代之后的结果，必将使这个网络的扩展性大大增强，使存储系统和主机系统可以远隔千里进行通信。

得益于 FC 带来的诸多好处，现在人们终于可以摆脱存储系统和主机必须放在一起的限制了。如果主机在北京，而盘阵可以在青岛，它们之间通过租用 ISP 的光缆线路进行连接，在这条线路上承载 FC 协议，从而达到主机和盘阵之间的通信。这样，在北京的主机上就可以直接认到远在青岛的盘阵上的 LUN 逻辑盘。

由于 FC 接口速度可以是 1Gb、2Gb、4Gb 甚至 8Gb，并且盘阵前端可以同时提供多个主机接口，所以它们带宽之和远远高于后端连接磁盘的并行 SCSI 总线提供的速度。这样，就可以在盘阵的后端增加更多的 SCSI 通道，以便接入更多的磁盘来饱和前端 FC 接口的速率。

9.2 不甘示弱——后端也升级换代为 FC

在将主机与盘阵之间的接口、链路都替换成 FC 协议之后，人们不断增加磁盘阵列后端磁盘的数量，以达到前端众多 FC 接口的饱和速率。但是此时瓶颈出现了，后端每增加一个

SCSI 通道，最多能接入 15 块磁盘，数量太少了。增加 SCSI 通道也不是一个最终解决办法，能否找到一种彻底地解决办法呢？

抛弃老爷车——让 SCSI 搭乘高速专列

要解决这个问题，就要彻底抛弃盘阵后端的 SCSI 并行传输总线网络，就像当初抛弃前端 SCSI 总线一样。而且不仅仅要使接入硬盘的数量增加，还要高速、稳定。既然已经将前端传输网络替换成了 FC 交换网络，那么是否可以将后端网络也从并行 SCSI 替换成 FC 呢？理论上是完全可行的。

由于 FC 协议系统提供了两种网络拓扑架构，所以要考察后端存储网络到底使用哪种架构比较合适。首先交换式架构 Fabric 是一种包交换网络，寻址容量大，交换速度快，各个节点间可以同时进行线速交换，无阻塞通信，但是成本较高。其次是 FC-AL 仲裁环架构，带宽共享，每个环寻址容量 128，最关键的一点，是它实现起来比交换架构简单，而且成本也低很多。第 7 章讲的 FC 的一些登录、注册等过程，都是在 Fabric 架构中才发生的，而在 FC-AL 则是另外一套环初始化过程。

如果把每个磁盘都作为一个节点连接到 Fabric 交换网络中，性能绝对是无可挑剔的，由共享总线变成了点对点交换式通信，性能也会提升很大。但是这样做，不但要在盘阵的后端实现一个 FC 交换矩阵，而且要在每块磁盘上实现 FC 拓扑中的 N 端口，这两部分成本是非常巨大的。为了降低成本，只能选择性能稍差，但是成本低的 FC-AL 仲裁环架构来连接磁盘阵列的控制器和磁盘，而且在每块磁盘上都实现 FC 拓扑中的 L 端口。

虽然这样做性能会比 Fabric 架构差，但是至少比并行 SCSI 总线强多了。对于并行 SCSI 总线来说，目前最高的标准是 Ultra 320，裸速率 320MB/s，实际最大传输率大概能有 85% 的效率，也就是 280MB/s 左右。有人做过实验，在 Ultra 160(裸速率 160MB/s)的总线上，按 4KB 数据块随机访问 6 块 SCSI 硬盘时，SCSI 总线的实际访问速度为 2.74MB/s，IOPS 大约 700 次/s。这种情况下，SCSI 总线的工作效率仅为总线带宽的 1.7%。在完全不变的条件下，按 256KB 的数据块对硬盘进行顺序读写，SCSI 总线的实际访问速度为 141.2MB/s，IOPS 大约 564 次/s，SCSI 总线的工作效率高达总线带宽的 88％。

由于 FC-AL 目前刚刚普及到 4Gb 的带宽，裸速率 400MB/s，这样就比 Ultra 320 的带宽要高。不仅在速率上，FC 在效率上也比 Ultra320 并行总线要高。但是 2G 速率的 FC-AL，其裸速率仅 200MB/s，比 Ultra 320 低很多。而在 4G 的 FC-AL 出来之前，很多磁盘阵列就已经用 2G 的 FC-AL 替代后端的 Ultra 320 总线了。为何这些产品宁愿忍受 2G 速率 FC 相对 Ultra 320 一半的速度，也要将其后端替换为 FC 架构呢？

其因素主要有三个。

- 可扩展性。受并行 SCSI 总线仲裁机制本身的限制，决定了一条总线上不会有太多的节点，16 个节点的数量已经达到它的可管理极限了。而 FC-AL 仲裁环则不然，它的极限是 128 节点，这就比 SCSI 总线强多了。这个限制的突破，使后端可以连接更多的磁盘，很容易就可以在单台磁盘阵列上实现上 TB 或者几十 TB 的容量(通过连接扩展柜)。

  虽然一个 FC-AL 环的速度比一条 Ultra 320 总线低，但是多条总线和多个后端通道可以集成在一个控制器上。而并行 SCSI 接口和线缆都很宽大，想集成在一个小的空间上很难。FC 由于是串行传输，两条线一收一发足够了。接口也很小，如 SFP 光纤接口，只有指头肚一样大小，可以很方便地在后端上实现多个通道。

- IOPS 值比并行 SCSI 总线的架构显著增加。为什么这么说呢？我们分析一下。高 IOPS 通常意味着 IO SIZE 值比较小的情况下。如果使用并行 SCSI 总线，由于可接入节点数量较少，磁盘数量少，每秒可接受的 IO 请求就少。而 FC-AL 的后端，一个通道可以连接 120 多个磁盘，可以做成很多 Raid Group，每秒可接受的 IO 请求就比并行 SCSI 多得多。所以，虽然 2Gb 带宽的 FC 网络传输在持续传输速率上比不过 Ultra 320 的 320MB/s 的速度，但是 IOPS 却比 Ultra 320 总线高很多。在特定条件下，2Gb 的 FCAL 链路会在 IOPS 和吞吐带宽指标上都超越 Ultra320 的 SCSI 总线。

  现在的大部分应用都是要求高 IOPS 的，它们产生的一个 IO SIZE 一般都比较小，而且随机 IO 居多。但是对于视频编辑等领域，无疑是要求高传输带宽的。面对这种应用，可以通过在盘阵后端加入多个 FC-AL 环来解决，后端带宽总和等于环数乘以环带宽。
- 双逻辑端口冗余。由于 FC 的串行方式使得数据针脚数量降低，相同的空间内很容易做成双逻辑端口冗余。双逻辑端口磁盘可以有效保证当其中一个端口发生故障之后，磁盘可以继续使用另外一个备用端口接受 IO 请求。

9.3 FC 革命——完整的盘阵解决方案

FC 在盘阵的前端接口技术的革命成功之后，又在后端的接口技术上取得了成功。FC 技术的两种拓扑，一个称霸前端，一个称霸后端，在磁盘阵列领域发挥得淋漓尽致。

与此同时，磁盘生产厂家也在第一时间将 FC 协议中的 L 端口和 FC 硬件芯片做到了磁盘驱动器上，取代了传统的 SCSI 端口。同时，根据 FC 协议的规定编写了新的 Firmware，用于从 FC 数据帧中提取 SCSI 指令和数据，完成 FC 协议通信逻辑。



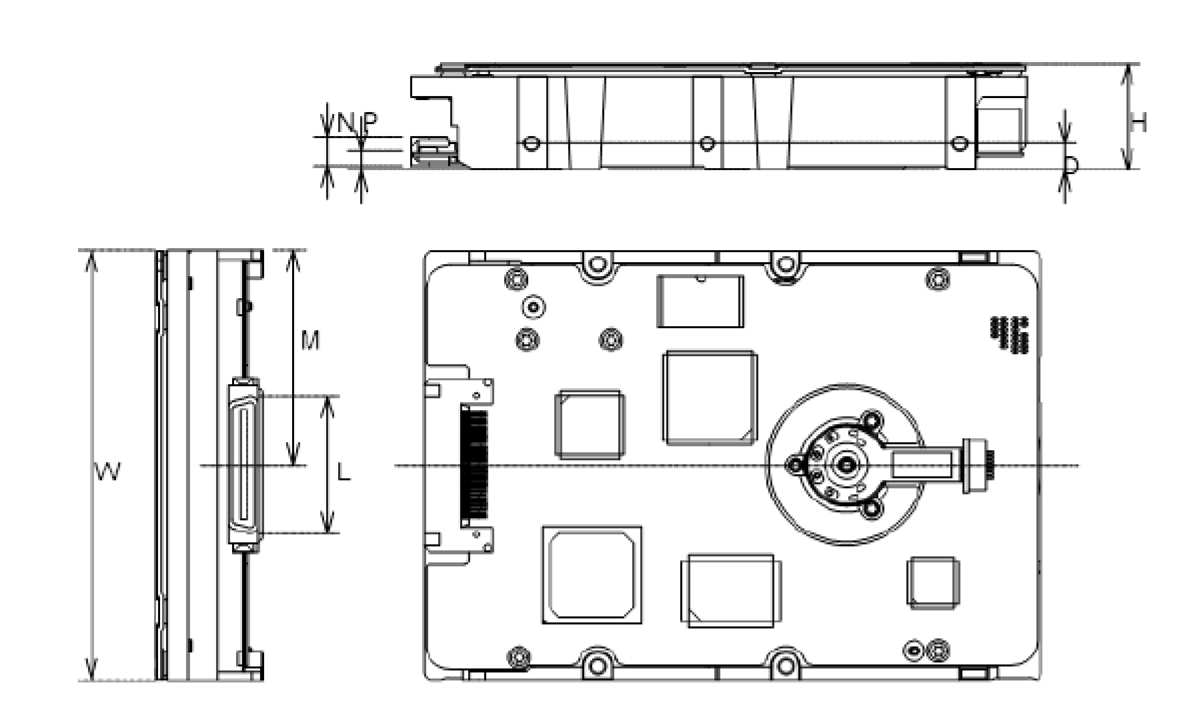
并行 SCSI 磁盘以及其他设备目前仍有比较广泛的应用，尤其是服务器本地磁盘。服务器本地磁盘一般只安装操作系统，一般情况下应用数据都会放到 SAN 的磁盘阵列上，所以对本地磁盘的性能要求不高，使用 Ultra 320 磁盘足矣。另外，普通独立磁带机一般也用 Ultra SCSI 320 作为其外部接口。只有在大型磁带库设备上，为了将其接入 FC SAN 才会使用 FC 接口。

因为要把每块磁盘都连接到 FC-AL 网络中，所以磁盘上要做上一个 FC 接口。由于磁盘阵列背板需要连接众多的磁盘，所以就注定不可能用柔软纤细的光纤来连接磁盘到背板，必须使用硬质铜线，让磁盘的 FC 接口用铜线来接触盘阵背板上的电路，这样才能做到方便地插拔。

9.3.1 FC 磁盘接口结构

FC 磁盘的接口为 SCA2 形式的 40 针插口，如图 9.1 所示。

SCA-2 Fibre Channel



Dimension	mm	Dimension	mm
H	26.1 max.	W	101.6±0.25
D	6.35±0.25	L	(30)
N	(7)	M	50.8±0.5
P	4.6±0.5		

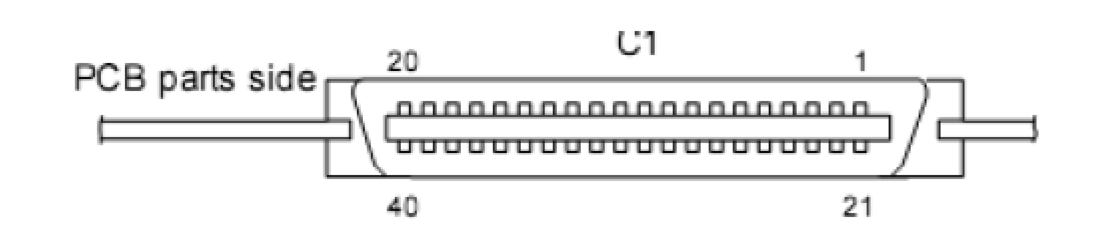


图9.1　SCA2 接口规格

从图 9.1 中可以看出，FC 磁盘的接口与 Ultra 320 SCSI 磁盘的接口形状完全相同。但是在接口上的物理信号定义和承载的上层协议是完全不同的，FC 磁盘接口是承载的 FC-AL 协议，而 SCSI 磁盘接口承载的是并行 SCSI 总线协议。图 9.2 是 FC SCA2 型接口的信号定义表。

Signal Name	Connector contact No.		Signal Name
-EN Bypass Port 1	1	21	12V
12V	2	22	GND (12V Return)
12V	3	23	GND (12V Return)
12V	4	24	+IN1
-Parallel ESI	5	25	-IN1
GND	6	26	GND (12V Return)
ACTLED	7	27	+IN2
Reserved	8	28	-IN2
START1	9	29	GND (12V Return)
START2	10	30	+OUT1
-EN Bypass Port 2	11	31	-OUT1
SEL6	12	32	GND (5V Return)
SEL5	13	33	+OUT2
SEL4	14	34	-OUT2
SEL3	15	35	GND (5V Return)
FLTLED	16	36	SEL2
DEVCTRL2	17	37	SEL1
DEVCTRL1	18	38	SEL0
5V	19	39	DEVCTRL0
5V	20	40	5V

图9.2　FC 接口 SCA2 针脚信号定义表

9.3.2　一个磁盘同时连入两个控制器的 Loop 中

图 9.3 和图 9.4 分别给出了一串磁盘以单 Loop 和双 Loop 接入的情形。

从图中可以看出来，原来这个物理接口中共包含了两套逻辑接口，可以分别接入一个 FC-AL 的环路中。

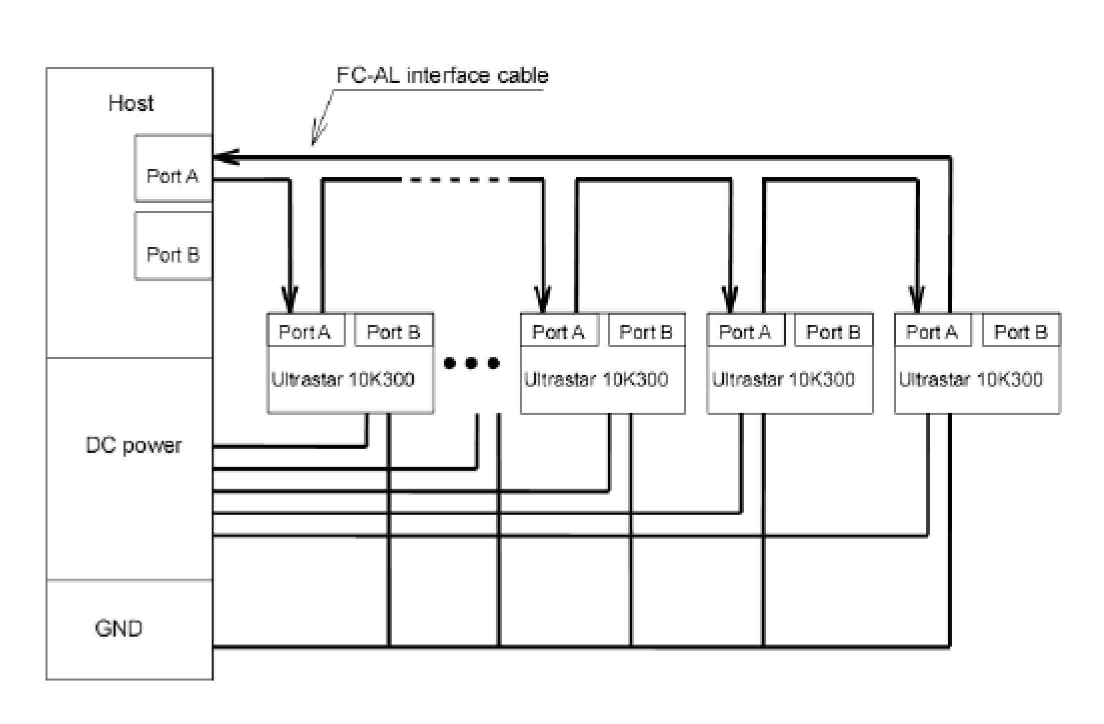


图9.3　单 Loop 连接示意图

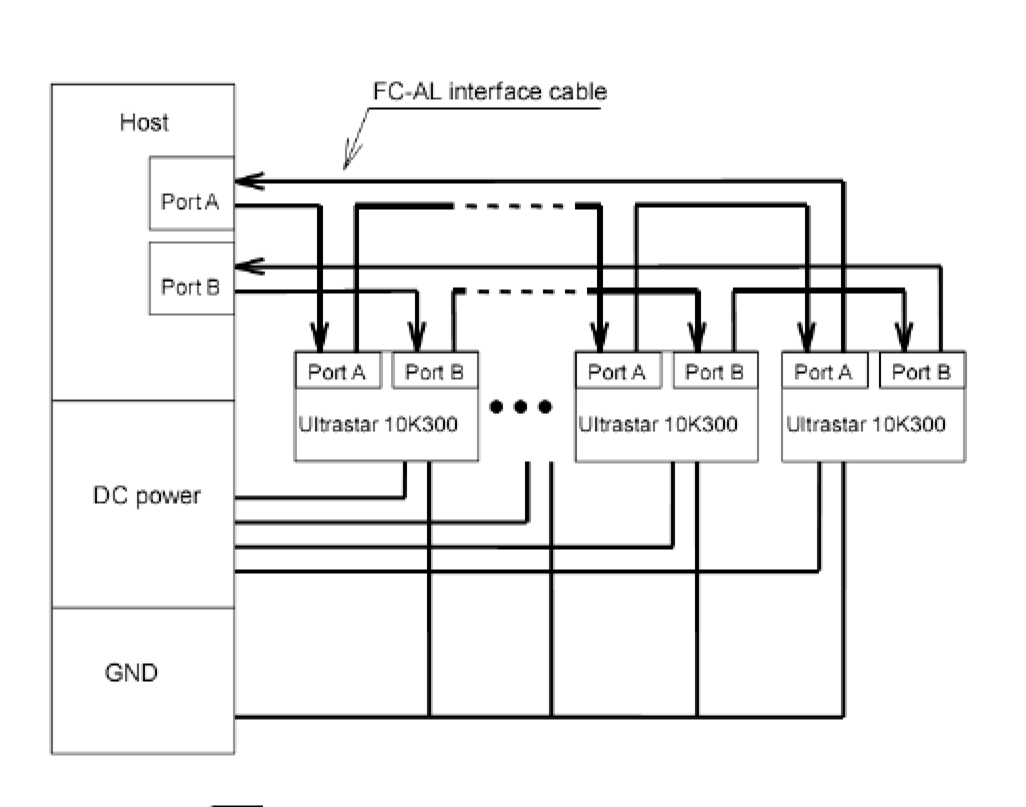


图9.4　双 Loop 连接示意图

9.3.3　共享环路还是交换——SBOD 芯片级详解

革命之后，被并行 SCSI 总线技术禁锢多年的存储系统终于解放了，迎来了全 FC 架构的春天。

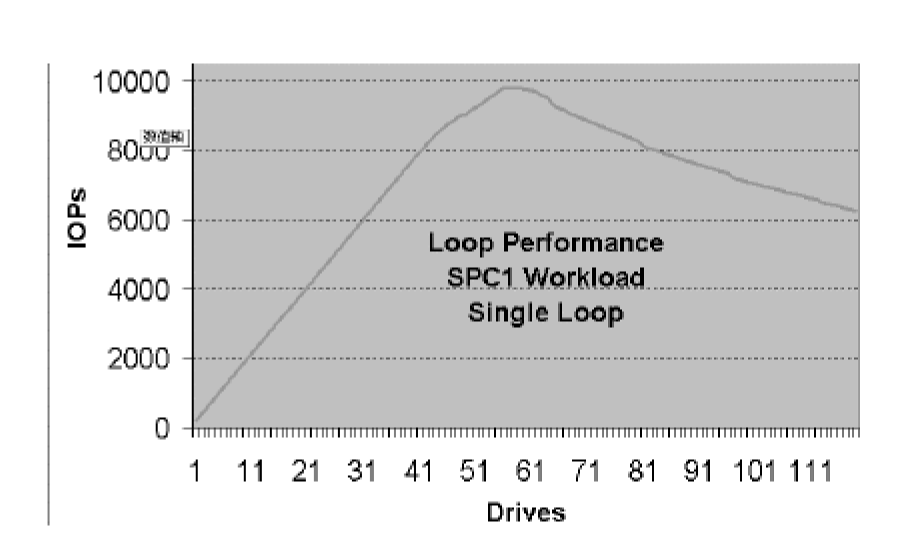


图9.5　FC AL 环的性能曲线

各个厂家分别推出了自己的产品。由于虽然一条 FC-AL 环最多能接 120 多块磁盘，但是有人测试过，环上节点的数量在最大值的二分之一时，性能达到最大化。再增加节点数量，性能不升反降。究其原因，可能是因为 60 个左右的节点，已经达到 FC-AL 环的仲裁性能以及带宽共享限制的极限了，如图 9.5 所示。如果再增加节点，那么用于仲裁所耗费的资源，就会影响性能的发挥了。这也是仲裁环或者总线的通病。对于 Fabric 架构，就没有这种限制。

思考

节点数量和仲裁/帧转发效率是一对矛盾系统，只能在二者之间进行折中选择。在 60 节点数量左右，能达到最大 IOPS。如果还想增加节点数量，增加盘阵所提供的总容量，那就只能牺牲 IOPS 性能。同样，每个环上的磁盘数量也不能太少，太少的话将达不到最高 IOPS，虽然此时仲裁和帧转发速度快。

有没有可能将后端的共享 FC-AL 环路架构，改变为交换式架构呢？改为交换式架构是可以，但是不能改为 Fabric，因为其成本相对偏高，而且 Fabric 的一些特性对于后端来说是用不到的。

Emulex 公司发布的 InSpeed SOC422 芯片、PMC-Sierra 公司的 PM8378 芯片等就是运行 FC-AL 协议但物理架构是交换架构的芯片。然而这个交换架构绝非 Fabric，因为其遵循的上层逻辑依然是 FC-AL 逻辑，只是在物理连接上用点对点交换架构，替代了“节点大串联”的 Loop 结构，使节点与节点之间传输的数据可以通过交换矩阵直达，而不是在环路上一跳一跳的中继。然而，这些芯片依然可以用在 Fabric 交换机上，只要经过一定改造并且在上层运行对应的 Firmware 即可。

其实就是用星型连接取代串联，而电路运行的逻辑依然是 FC-AL 仲裁过程，因为位于

控制器上的 FC 适配器依然会执行 FC-AL 仲裁等逻辑，只不过这个仲裁过程变得非常简单，不再需要所有磁盘参与，而由这块芯片来进行仲裁。此外，某节点同一时刻依然只能与一个节点通信，节点感觉不到底层电路架构的变化。由于同一时刻还是只能存在一对节点进行通信，所以链路带宽依然是共享的。因此，这种交换架构做的并不彻底，它没有过渡到包交换或者所有端口无阻塞全交换架构，虽然物理上已经可以实现点对点的通路。

图 9.6 所示的为传统 FC-AL 架构和半交换式 FC-AL 架构示意图。我们可以看到左边的拓扑完全就是磁盘串，而右边则变成了星型的架构，中间由一个 ESH 交换模块辐射出来。

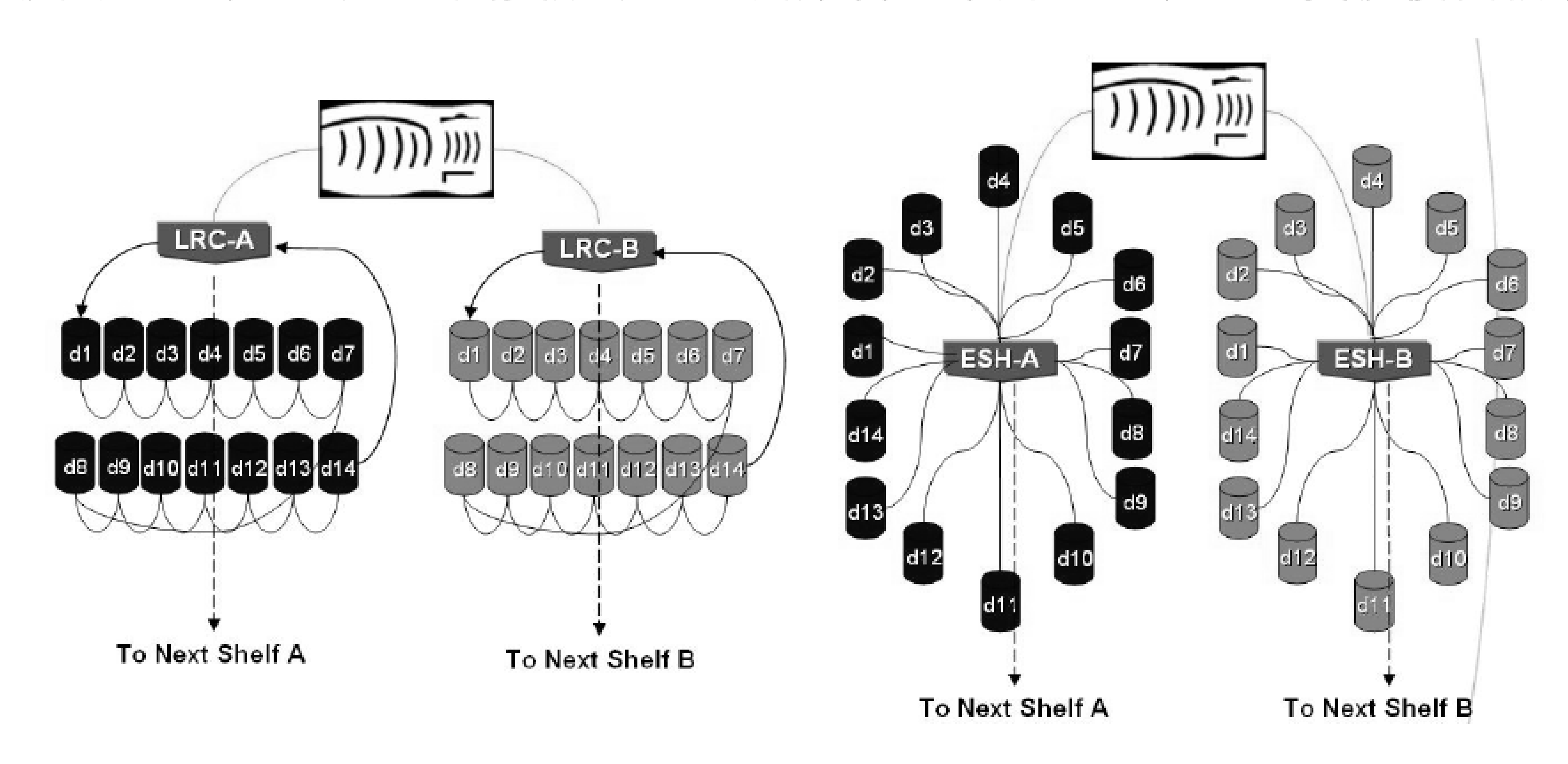


图 9.6　传统 FC AL 环和半交换式 FC AL 环

目前高端盘阵后端控制芯片几乎全部采用半交换式架构了。半交换式架构有以下好处。

- 控制节点与所有其他节点间都是点对点矩阵直连架构，相对于环路架构减小了数据传输时的延迟。数据帧不需要一跳一跳的转发，可以直接从发起者到达目标。
- 可以快速侦测和隔离某个节点的故障而不影响其他节点。
- 由于降低了链路延迟，增加了效率，所以在性能可接受的前提下，一条链路可接入的节点数大大增多。
- 相对于纯环路架构，半交换架构提高了传输速度和 IOPS。

纯环路架构的扩展柜中，连接硬盘的背板只是一个 FC-AL 环路连接装置，而升级到半交换架构的硬盘扩展柜的，其连接硬盘的背板上就有了 Switch 芯片(其实这个芯片一般存在于从背板单独接出扩展模块上)，可以级联多个扩展柜。在逻辑上，这些级联扩展柜中的所有磁盘属于一个逻辑上的 Loop，前者称为 JBOD(Just Bunch Of Disks)，后者称为 SBOD(Switched Bunch Of Disks)。关于这些模块的实物图，请参照本书第 6 章的相关章节。

1. PMC-Sierra 公司 PM 系列芯片简介

下面以 PMC-Sierra 公司的 PM8368 和 PM8378 两款芯片来详细解释一下这种芯片的作用方式。

(1)　PBC 芯片。

在传统模式的 FC-AL 架构下，所有环路上的节点(adapter 和磁盘)都通过串联架构串接到了一起。针对这个架构的 PM8368 芯片是一款具有 18 个 2Gb/s 的 FC 接口的带有 PBC(Port Bypass Control)功能的芯片，相当于一个 Loop 串联器，并且带有端口 Bypass 功能，可防止某个端口故障引发的全 Loop 断开。图 9.7 所示为 PM8368 芯片的方框图。其 18 个端口

中，通常用 16 个连接磁盘，其他两个连接上位芯片和(或)下位芯片。这种不带有交换逻辑的普通 Loop 芯片，业界一般称为 PBC 芯片，即旁路控制芯片。

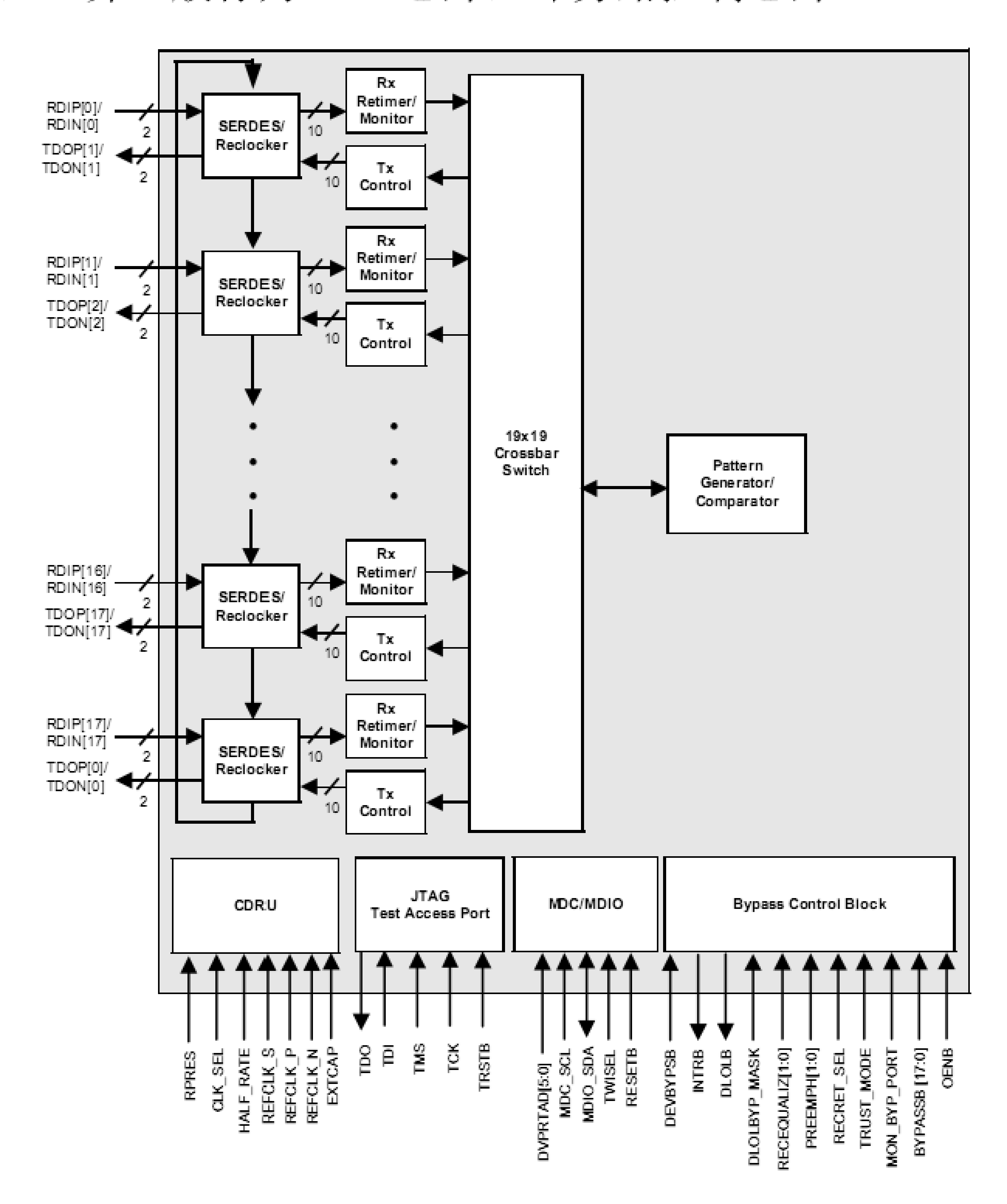


图9.7　PM8368 芯片方框图

图 9.8 所示的是将 PM8368 芯片与 PM8372 芯片(一款只有 4 个 FC 口的 PBC 环路集线芯片)搭配在一起而实现的扩展柜级联，每扩展柜有 16 块盘。引入 PM8372 芯片的原因是将其作为一个二级级联桥来减缓一级芯片(PM8368)的端口耗尽问题。一级芯片需要连接最终硬盘等设备。

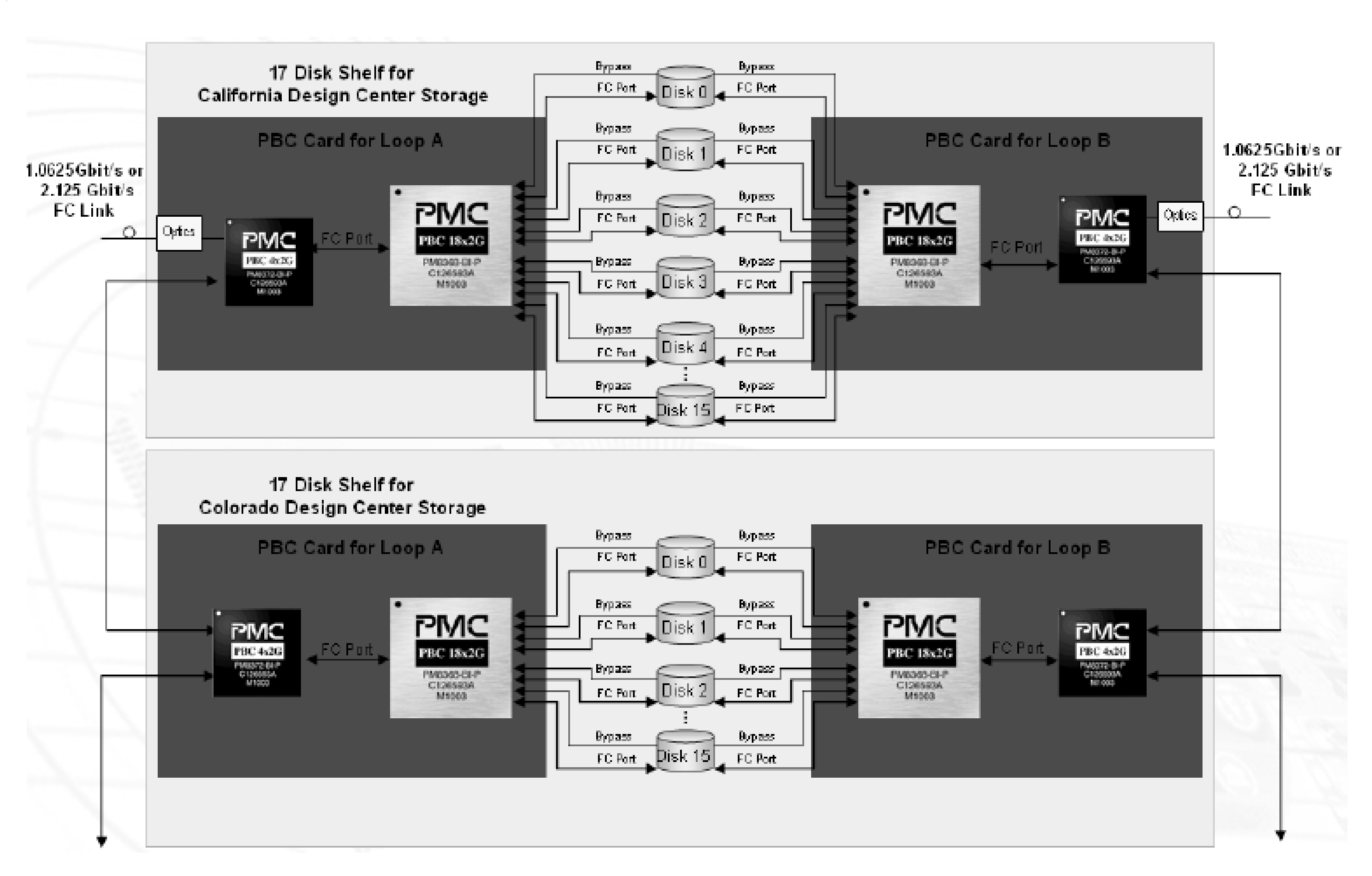


图9.8　利用 PM8368 芯片与 PM8372 芯片级联扩展柜

(2) CTS 芯片。

图 9.9 所示为 PM8378 芯片的方框图。此芯片为新一代的 CTS 芯片，CTS 为 Cut Through Switch 的简称。这款芯片具有交换逻辑，IO 性能相对于 PBC 芯片大大增加，而延迟大大降低。这块芯片可以接入 18 个 4Gb/s 的 FC 接口设备。在上部的模块中，可以看到三个模块：Arb Mgmt(仲裁管理模块)、AL_PA Table(AL-PA 地址端口映射表模块)、Cut Thru Mgmt(捷径交换模块)。

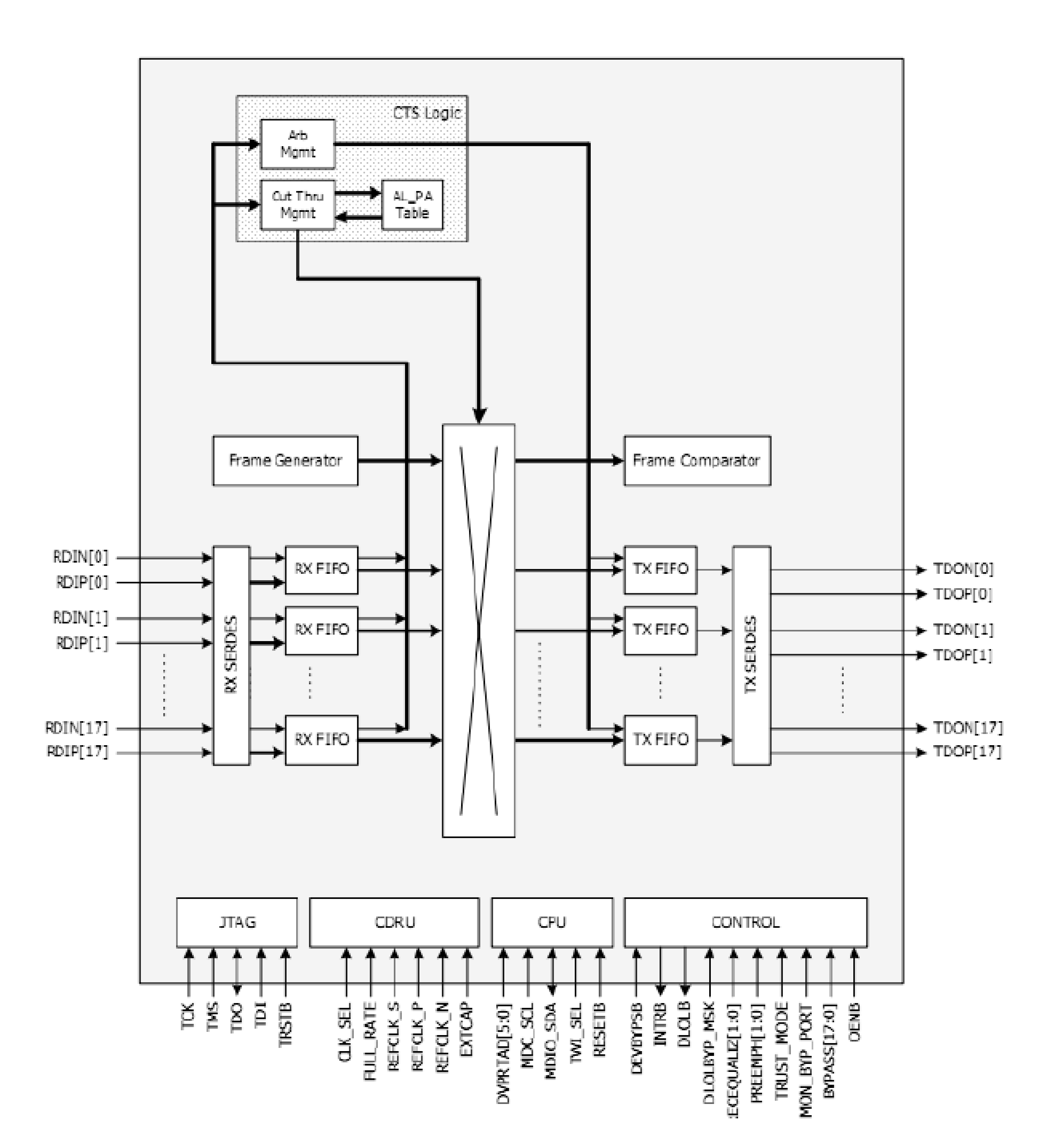


图9.9 PM8378 芯片的方框图

- Arb Mgmt(仲裁管理模块)：前面曾经说过，一个交换芯片必须感知 FCAL 的仲裁逻辑，才能作为一个中心仲裁器总揽仲裁权，而不用 Loop 上所有的节点都参与进来。这个仲裁管理模块实现的就是这个功能。FC 适配器只要发出仲裁请求，芯片就会马上通过，适配器立即就获得了使用权，加快了仲裁的速度。如果某时刻多个设备都发起仲裁，则芯片会根据自己的逻辑来处理这些请求，比如根据优先级等。
- AL_PA Table(AL-PA 地址端口映射表模块)：传统的 FCAL 环路上，数据从通信源被发送到对应 AL-PA 地址的目的设备，数据帧需要一站一站的向下接力传送。交换芯片的另一个作用就是抛弃这种低效的传输方式，使数据可以一站直达目的。芯片使用一张映射表来维护交换逻辑，这就和以太网交换机维护一张 MAC-端口表一样。FCAL 交换芯片每收到一帧数据，就会根据这个 AL_PA Table 来查找帧中的 AL-PA 地址所对应的芯片针脚是哪一个，然后直接将此帧转发到对应的针脚上，也就传送到了针脚所连接的磁盘 FC 接口上。
- Cut Thru Mgmt(捷径交换模块)：这个模块其实就是执行交换过程的。每当有帧需要交换，这个模块就会发送一个信号到图 9.9 中部的那个矩阵上，改变矩阵当前的通路布局，从而将数据从源传向目的。

图 9.10 是利用 PM8378 芯片实现的磁盘扩展柜级联示意图。18 端口中有 16 个连接了磁盘，另外两个分别连接了上位和下位芯片。

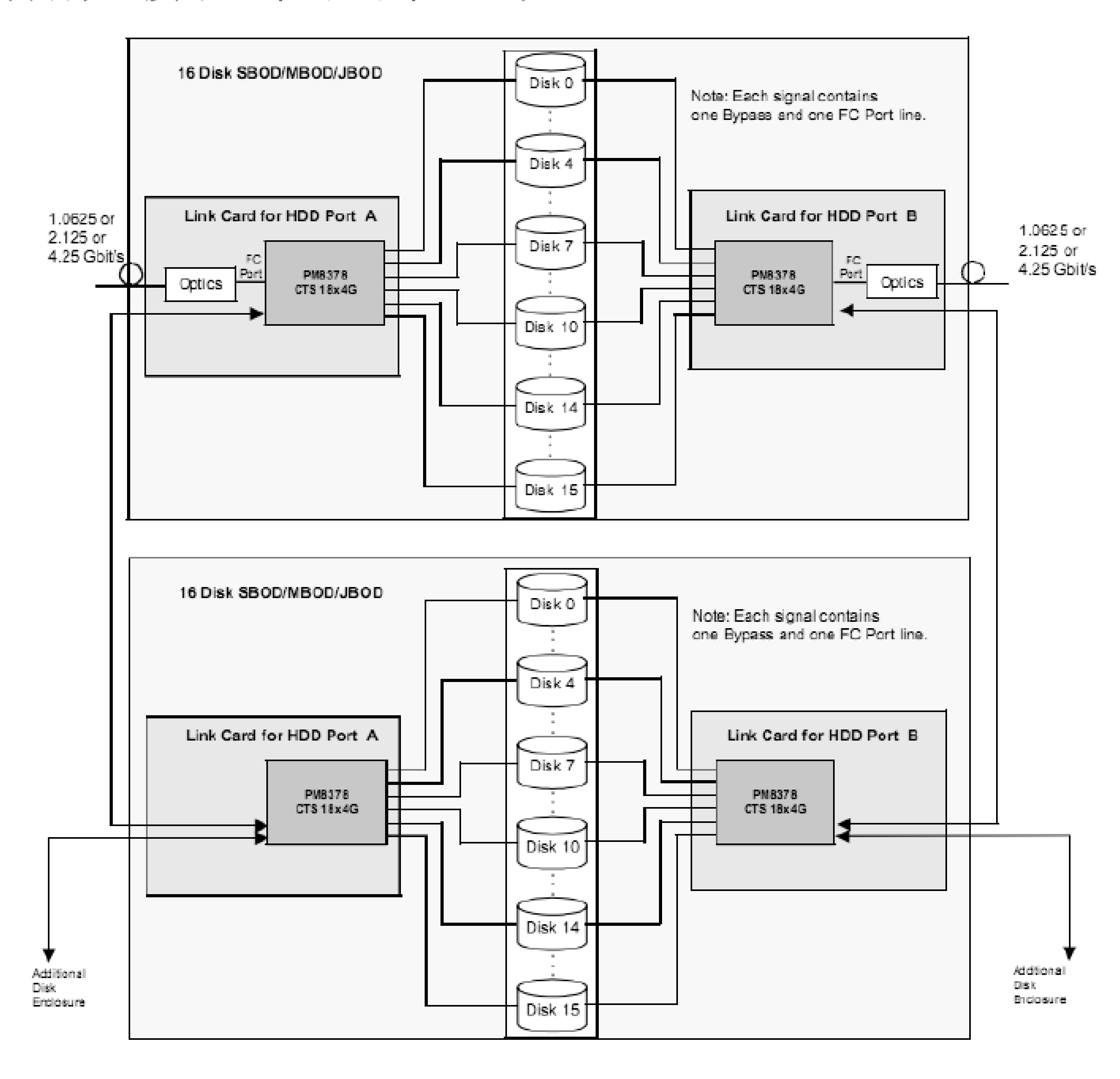


图 9.10 扩展柜级联示意图

还可以将 PBC 芯片和 CTS 芯片组合在一起来级联扩展柜。如图 9.11 所示，为了降低成本，使用了 18 口的 PBC 芯片来连接磁盘，然后用一个 4 端口的 CTS 芯片来做为二级级联桥。在这种情况下，每个扩展柜中的磁盘之间为普通 Loop 串联，但是扩展柜与扩展柜之间却是交换拓扑，这样就相当于把整个 Loop 上的节点划分成域，每个扩展柜就是一个共享域(冲突域)。

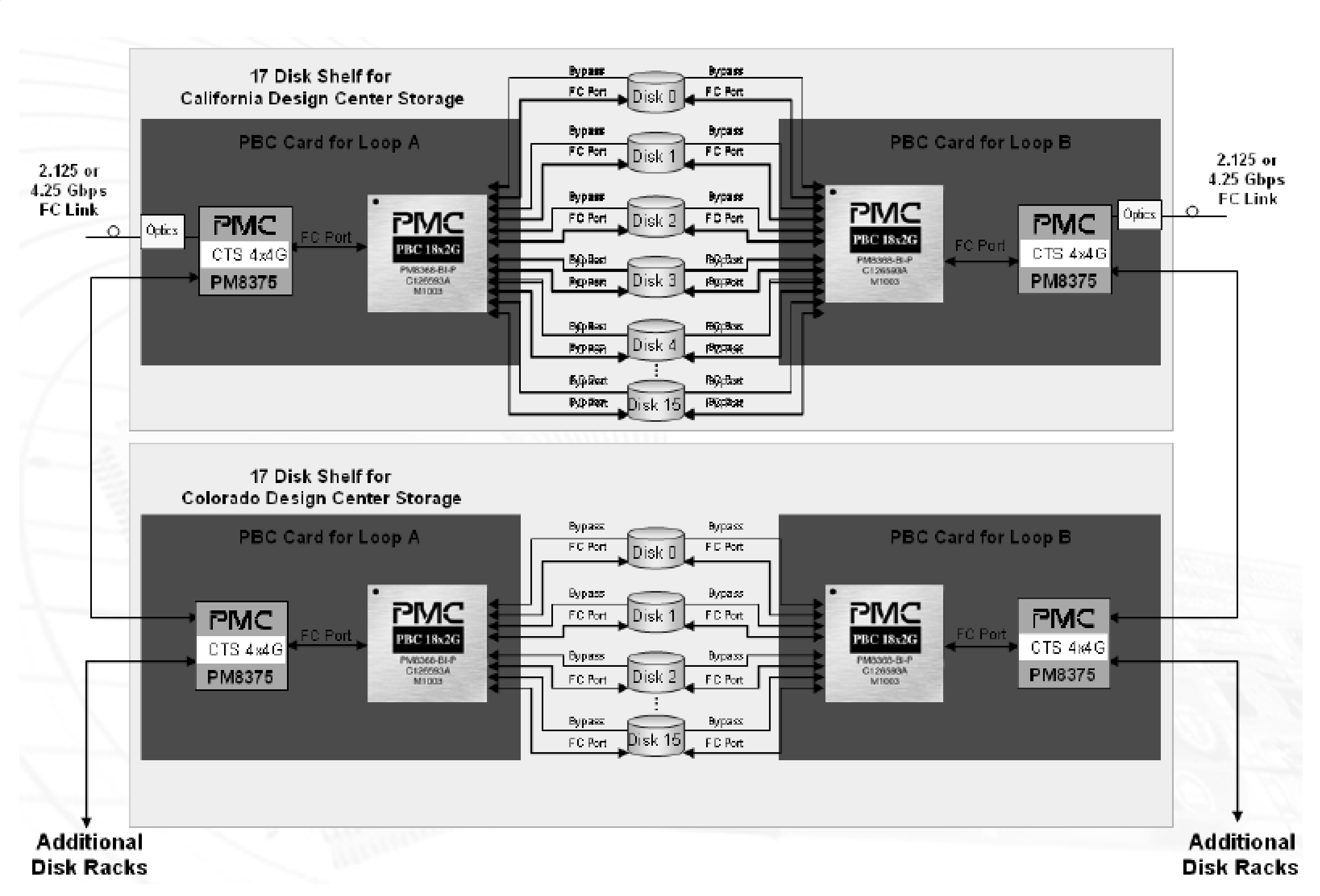


图 9.11 PBC 与 CTS 的组合

如图 9.12 所示，这种方式使用了高成本的 CTS 芯片组成了全交换架构的 Loop。整个 Loop 上的节点之间都形成了交换架构，但这个交换架构并不是无阻塞多路同时交换的架构，依然是同时只允许两点间通信的上层 FC-AL 逻辑的架构。

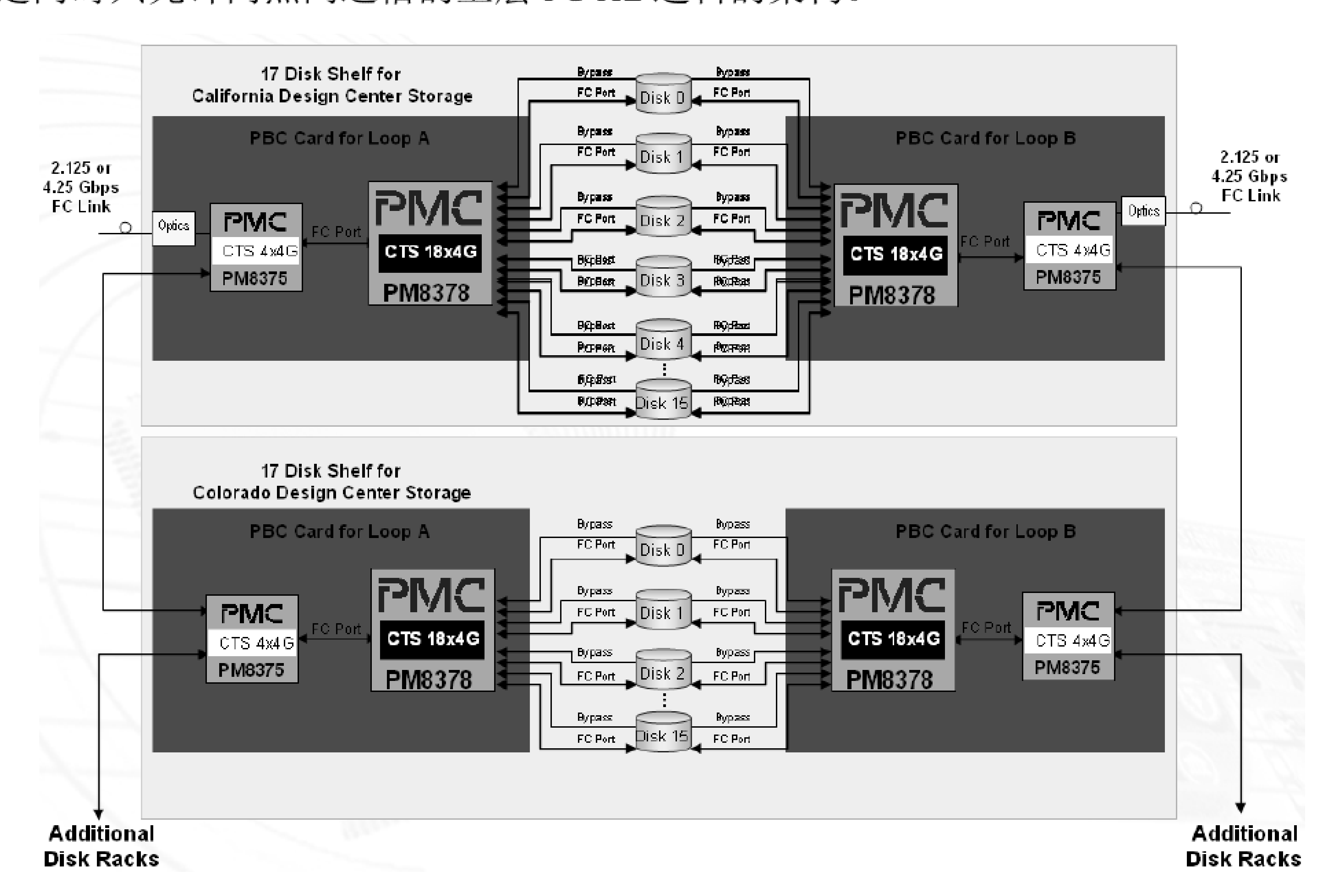


图9.12 全 CTS 架构

SBOD 控制模块上的芯片不只有 PBC 或 CTS，还包括其他各种功能的芯片，在图中并没有给出，在下面的内容中会一一介绍。

2. SCSI Enclosure Service 简介

SCSI Enclosure Service 简称为 SES，从字面上理解就是 SCSI 协议中用于查询磁盘扩展柜(Enclosure)各种状态的一种服务(协议)。扩展柜上有很多组件需要被监控，比如电源模块、风扇散热模块、各种指示灯、温度传感器等。这些组件都通过某种总线(比如 I2C、GPIO)方式连接到某个芯片，然后这块芯片再通过 I2C 等连接到单片机或 CPU。或者通过外部传感器直接连接到高集成度的单一芯片上。SCSI 客户端程序(也就是 SCSI Initiator 端程序)通过发送 SES 协议规定的各种指令来查询 Enclosure 上各个组件的状态信息。

SES 服务模块就是指扩展柜上的 CPU，其中运行着处理 SES 的代码。这个 CPU 可以是主控 CPU，也可以是独立的芯片。存储设备机头主控制器有两种方式将 SES 指令传输给这个 CPU，如下所述。

(1) 独立 SES 处理模块

如图 9.13 所示，独立的 SES 服务模块独占一个 LUN ID(十六进制的 0D)，可以直接寻址。也就是说，SES 服务模块就相当于一个 FCAL Loop 上的 ID，机头直接将 SES 指令传输给这个 ID。对应电路层面，主交换芯片收到目标为此 ID 的帧，便会直接转发给这个 ID 所连接的设备，这里就是 SES 处理芯片。

(2) 附属的 SES 服务模块

如图 9.14 所示，附属的 SES 服务模块往往会利用一个已经存在的设备的地址来与自己

共用。而物理接口上，这个设备一般会有一条旁路来连接到 SES 服务模块，比如目前 FC 磁盘普遍使用的 SCA2 接口中就有对应的 ESI(Enclosure Service Interface)接口来连接到 SES 服务模块上(CPU)。存储设备的主控机头首先将一条 SES 可用性探询指令发送到这个已经被磁盘占用的 ID，如果这个 ID 对应的槽位上有连接设备，且该设备支持转发 SES 帧，则该设备会返回一个确认帧，帧中携带有特定 Page 的值，主控机头收到之后便会分析出这个设备是否支持 SES 转发。机头随后发出纯 SES 指令，磁盘收到之后，会通过 ESI 接口将 SES 帧发送给 ESI 接口的对端，当然就是 SES 处理芯片了，芯片收到 SES 指令之后，便通过 I2C(或者其他类型总线)总线去查询外部传感器的信息，然后封装成 SES 返回帧，通过 ESI 接口发送至磁盘，磁盘再转发给主控机头。磁盘的 Firmware 必须支持转发 SES 帧。

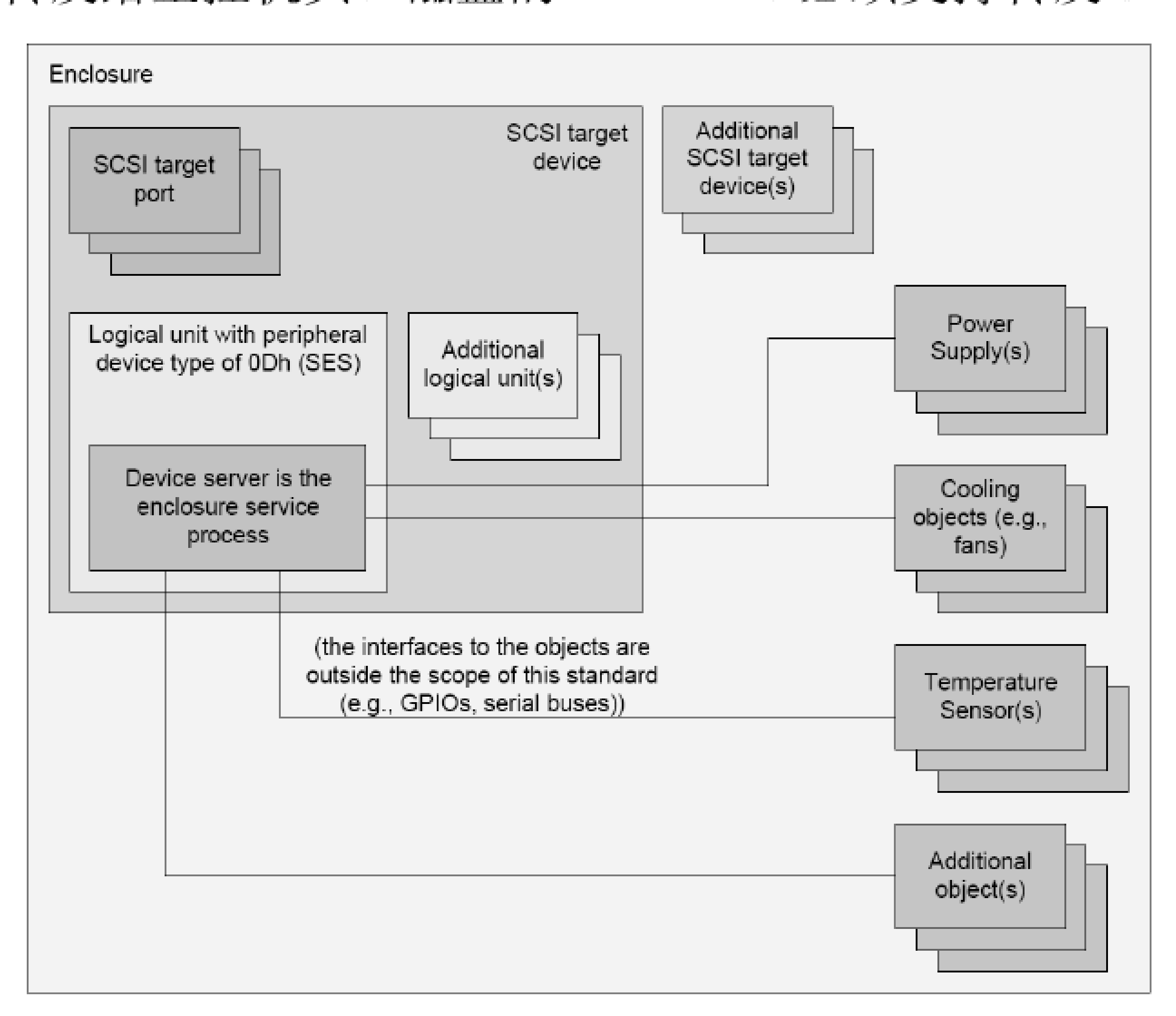


图9.13　独立 SES 服务模块

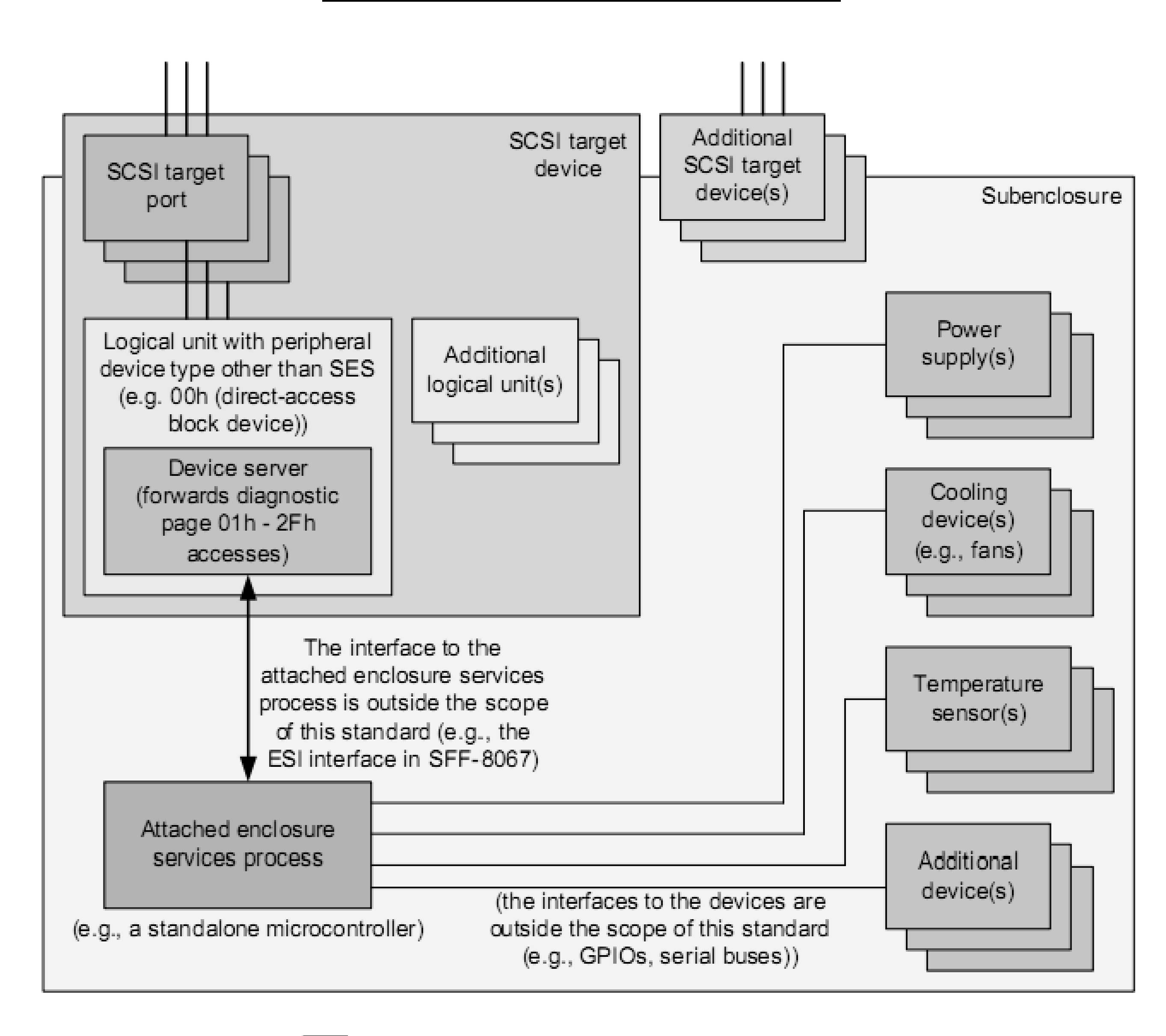


图9.14　附属的 SES 服务模块

3. SBOD 上的 CPU

(1) PMC－Sierra 公司的 PM8393 芯片。

这是一款基于 MIPS32 架构的单片机，内建有 128KB 的 RAM，外部时钟为 106.25 MHz，可以将其连接到上文所属的 PM8378 Switch 芯片中。这款芯片又名 Sotrage Management Controller，简称 SMC。图 9.15 为其方框图。

图 9.16 是 PM8393 芯片充当扩展柜总控 CPU，用 PM8378/PM8379 充当交换芯片来连接磁盘，再加上其他一些芯片共同组成的一个完整的扩展柜控制模块架构。

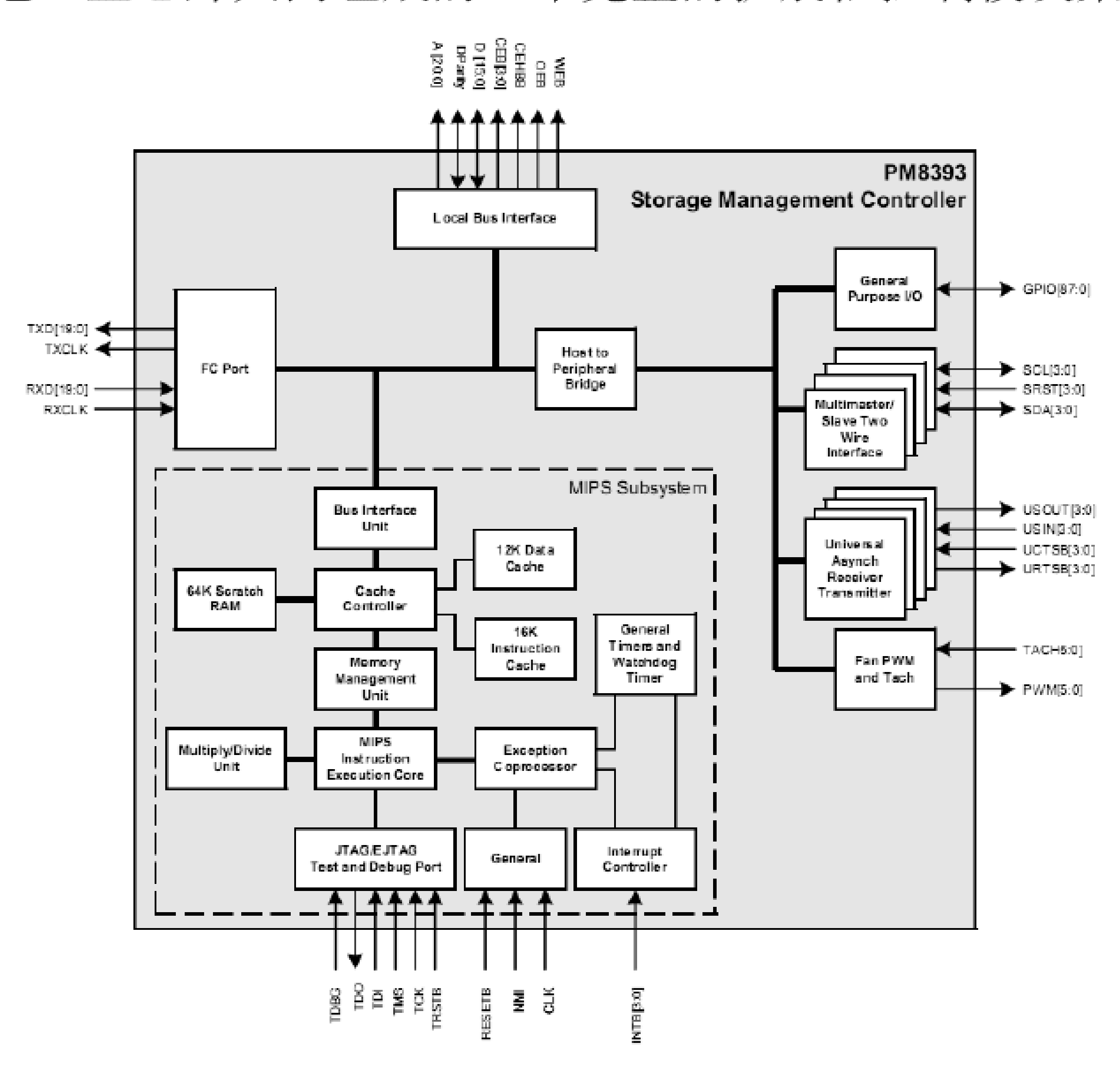


图9.15 PM8393 芯片方框图

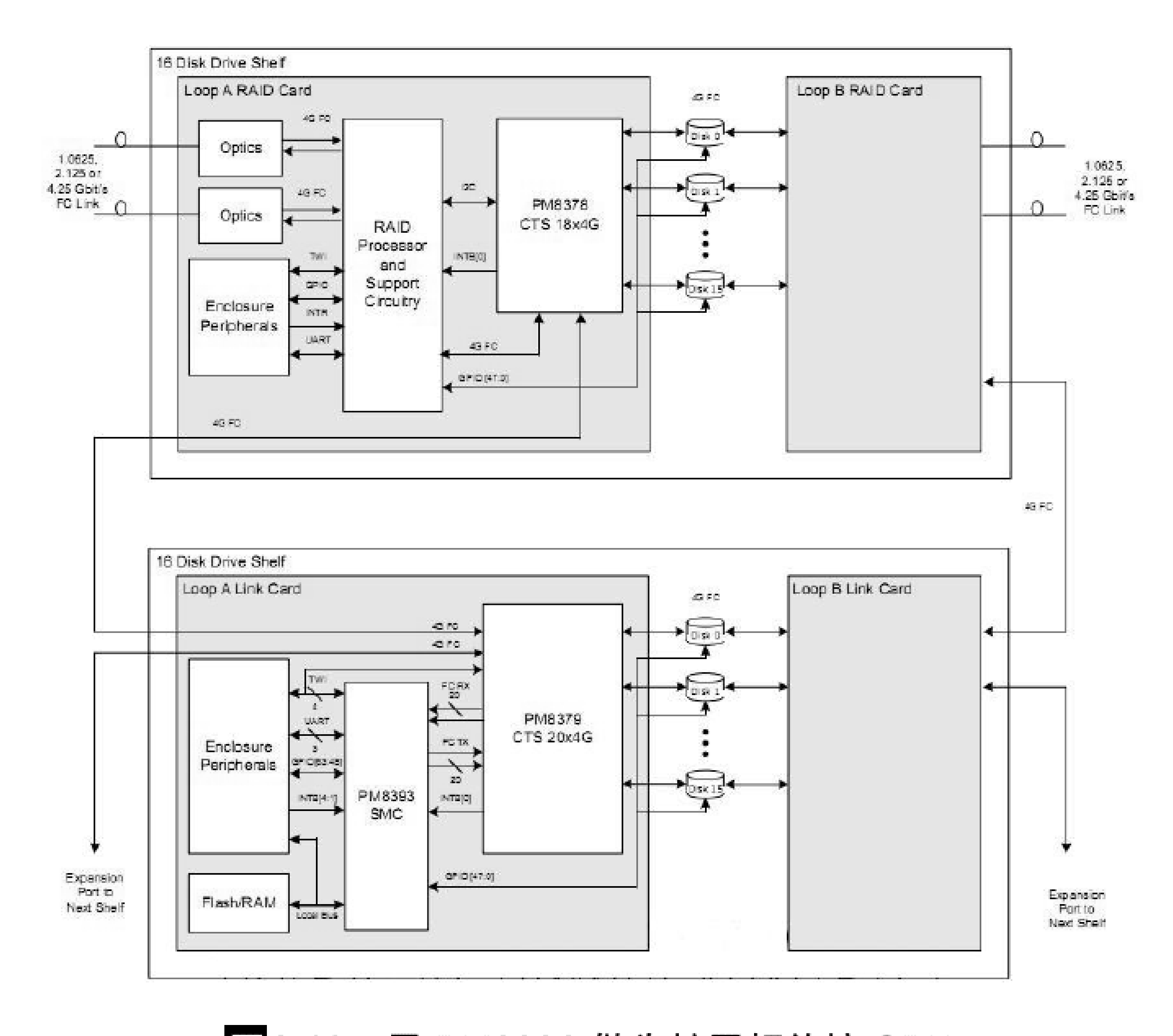


图9.16 用 PM8393 做为扩展柜总控 CPU

(2) Qlogic 公司的 GEM359 芯片。

图 9.17、图 9.18、图 9.19 所示为这款芯片的方框图。

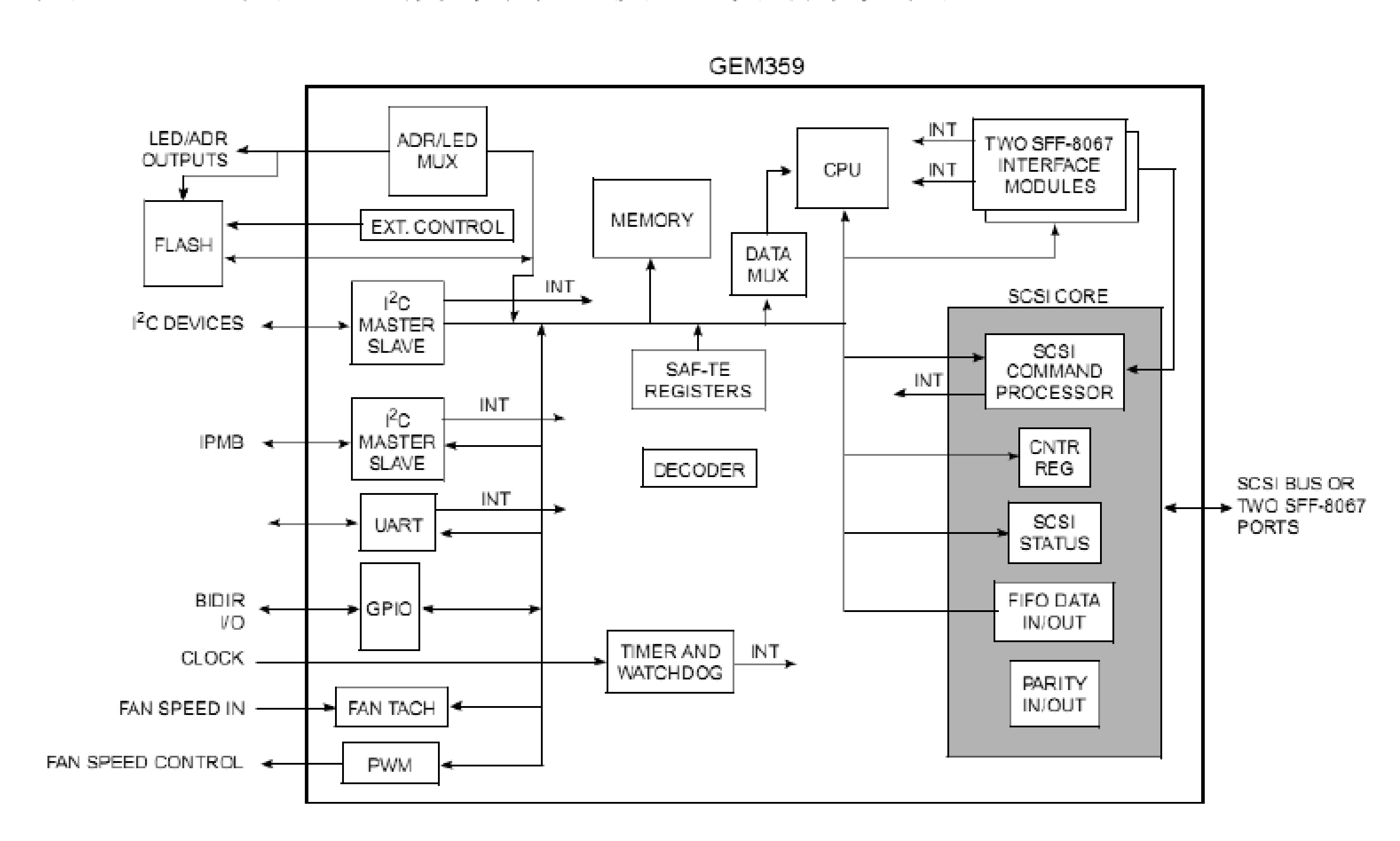


图9.17 GEM359 芯片方框图 1

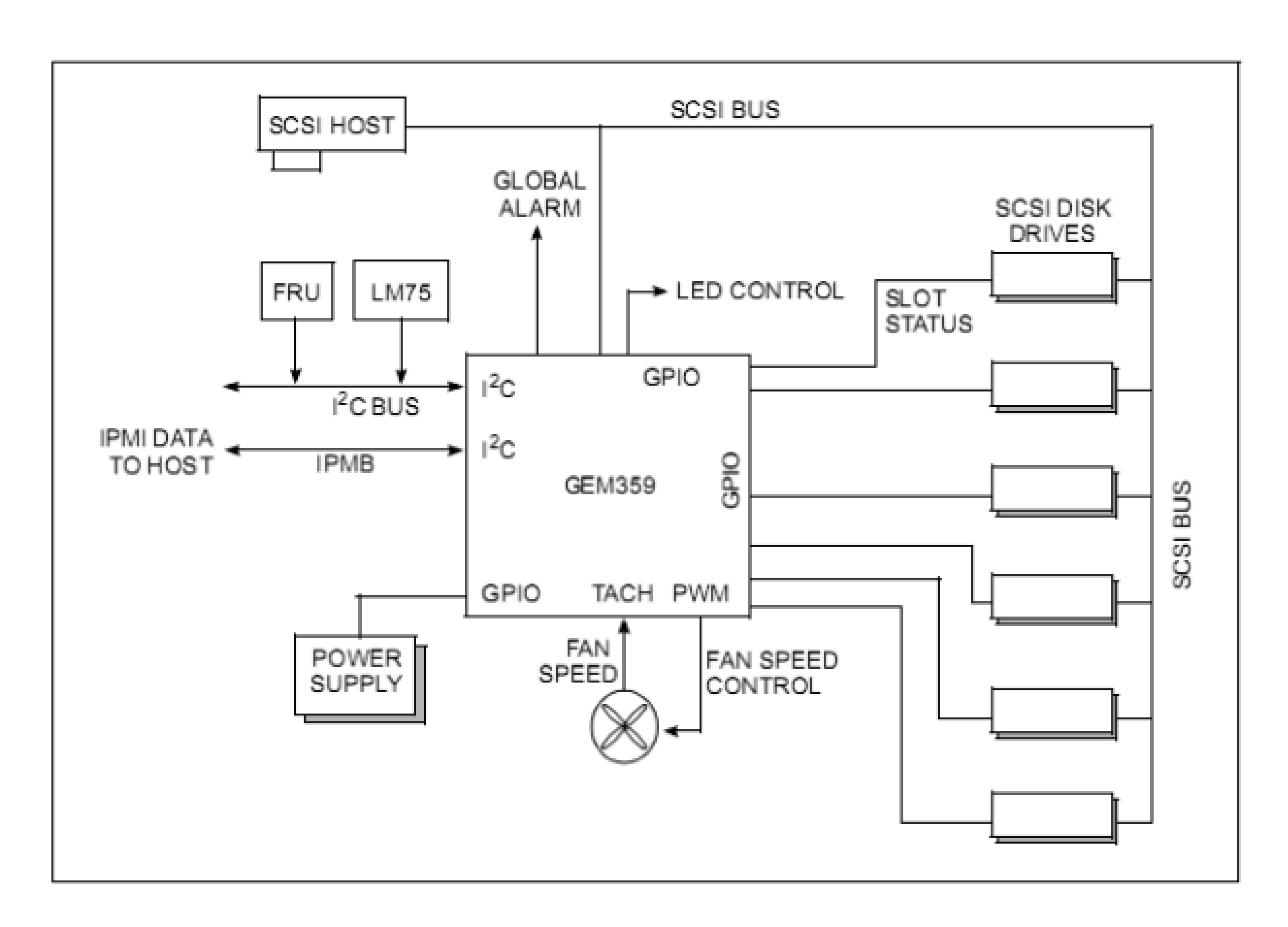


图9.18 GEM359 芯片方框图 2

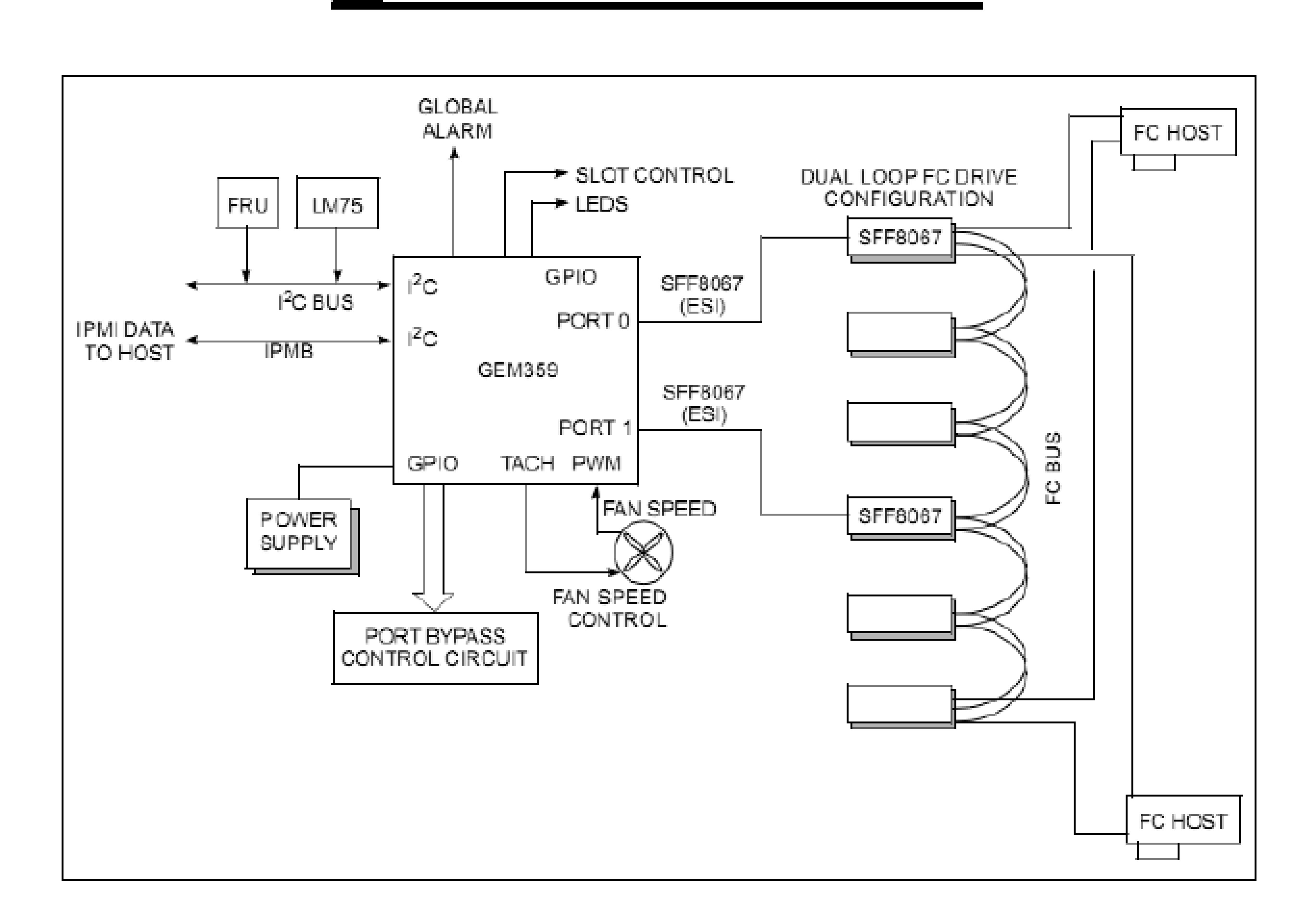


图9.19 GEM359 芯片方框图 3

4. SBOD 上的 ROM 和 RAM

SBOD 扩展柜上的 CPU 需要执行 Firmware 中的代码来完成一系列的动作，如 SES 服务等。所以，Firmware 编写的质量直接决定了扩展柜是否能向机头上的控制器稳定和迅速的报告扩展柜上所发生的一切事件。

Firmware 一般存在于扩展柜控制模块上的 ROM 芯片(如 Flash 芯片)中，并且可以随时将升级之后的 Firmware 通过 FC 接口直接写入芯片。这种直接通过实际数据链路来升级系统控制数据的方式叫做 in-band 升级，即“带内升级”。如果用一种单独的通道来访问 Flash 芯片并做升级动作就叫做 out-band 升级，也就是带外升级。

SBOD 控制模块上一般都有外置的 RAM 芯片，有些内置了 RAM 的单片机除外。

5. PATA、SATA 和 SAS 磁盘怎么办

PATA(IDE)盘和 SATA 盘相对于 FC 盘和 SAS 盘来说，成本降低了很多，且可以实现高容量，现在已经有了 1TB 的 SATA 磁盘。对于一些对 IO 性能要求不高的环境来说，使用 SATA 盘无疑是很合适的。但是面对不同的接口，不同的指令，单独对这些磁盘实现一套盘柜和控制器体系实在是不方便。且现在企业都要求高度整合，统一分配，方便管理。根据这个需求，各种适配器和转换逻辑出现了。图 9.20 所示为一个 SATA-SCA2 接口转换器。SATA 磁盘只要接上这块 PCB，就可以从物理上融入 FC 盘柜中，也就变成了所谓的 FATA 盘。除此之外，还有很多其他类型的转接电路，比如 PATA-SCA2、SAS-SCA2 等。

物理上融入了，在逻辑上也需要进行转换。SATA 磁盘使用 ATA 指令系统，FC 和 SAS 磁盘使用 SCSI 指令系统，二者不兼容。所以需要有一个中央芯片负责在两种逻辑之间互相转换。

Sierra Logic 的 SR1216 芯片是一款高集成度的芯片。这款芯片将 ATA-SCSI 转换逻辑以及两个 MicroProcessor 做到了一块单一的芯片中。MicroProcessor 就是微型 CPU，用来运行外部 Flash 芯片中的 Firmware，从而实现 SES 等扩展柜管理程序。

图 9.21 所示为 SR1216 实物图。

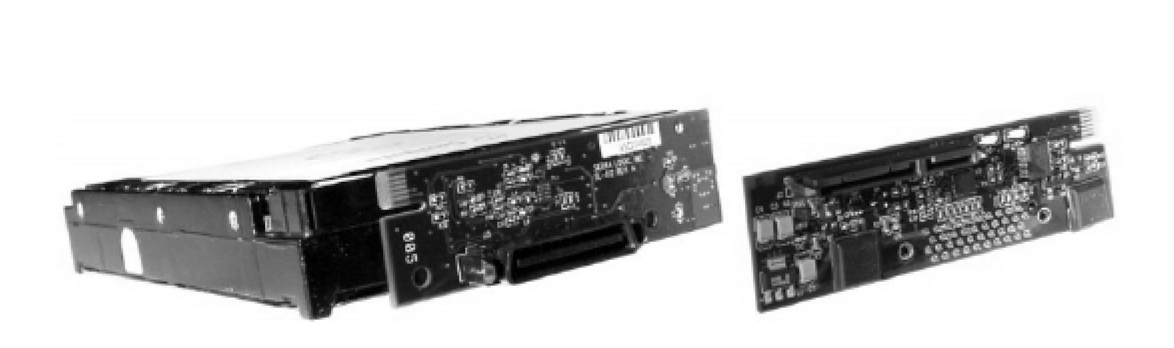


图9.20　SATA-SCA2 转接电路板



图9.21　SR1216 芯片实物图

图 9.22 和图 9.23 所示为用 SR1216 来充当 SATA 桥和总控 CPU 所形成的扩展柜控制模块架构图，PBC 表示 FC Loop 旁路控制芯片，SES Processor 表示用来收集外部传感器的专用芯片。

另外，PMC-Sierra 公司也有多款多端口 SATA 复用芯片，不过有一些需要搭配额外的 ATA-SCSI 转换桥芯片，图 9.24 为其示意图。

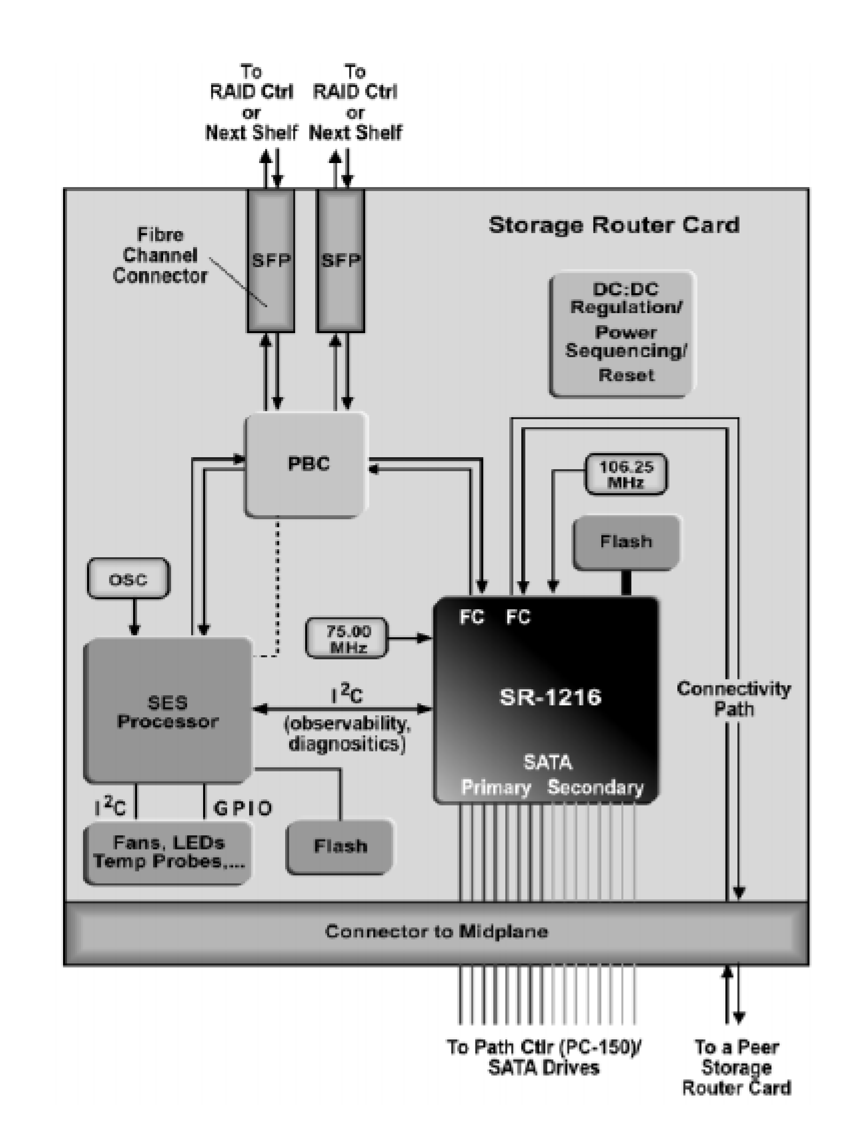


图9.22　SR1216 架构的扩展柜控制模块(1)

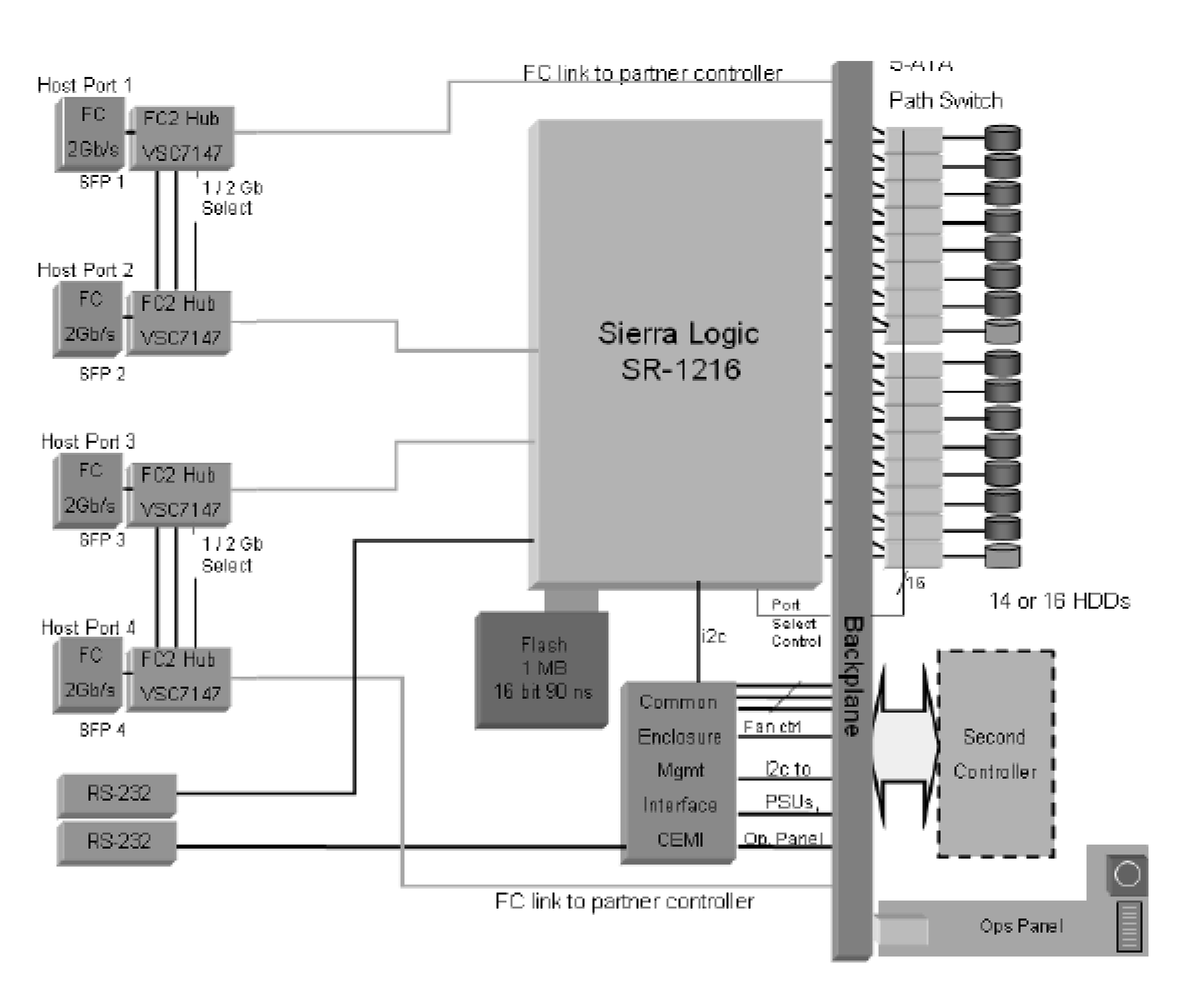


图9.23　SR1216 架构的扩展柜控制模块(2)

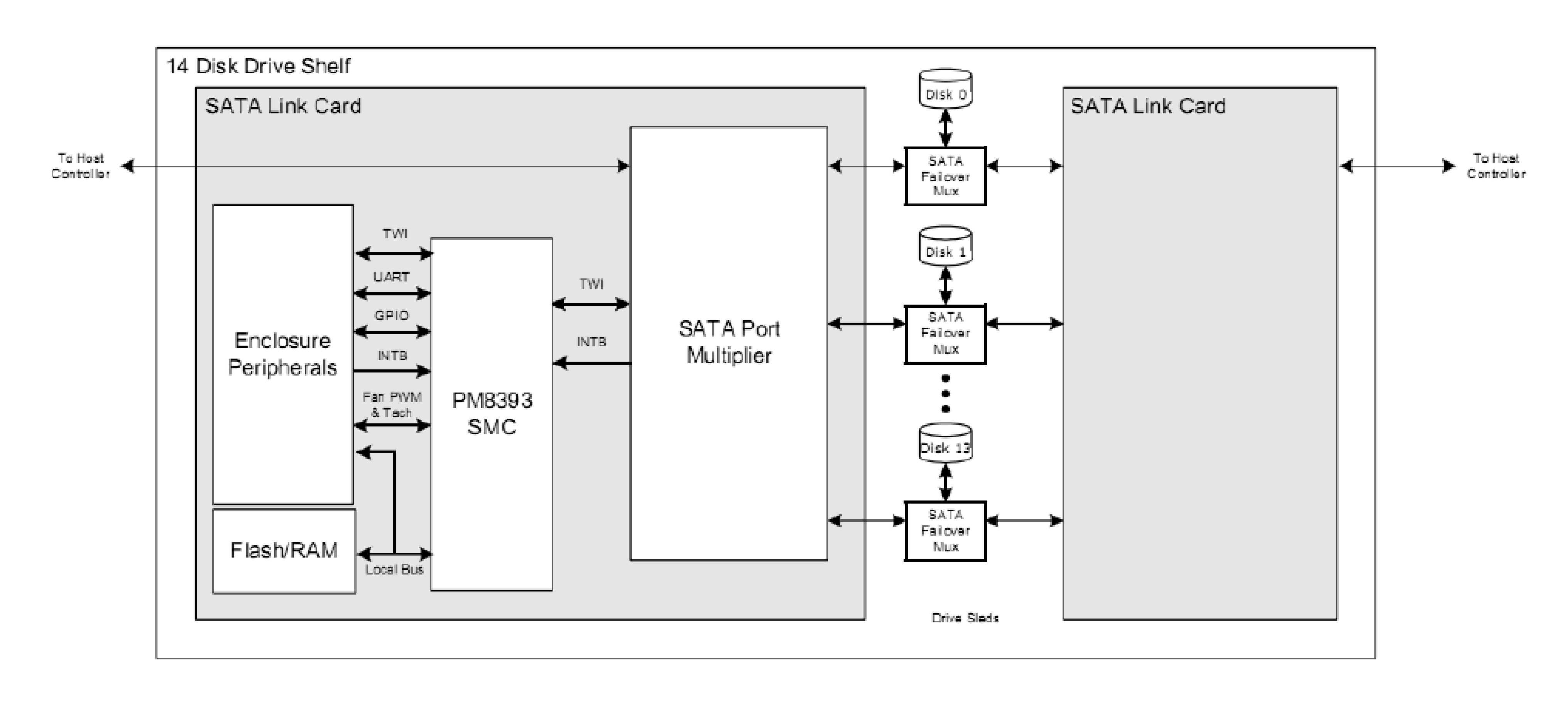


图9.24　PMC-Sierra 的 SATA 盘柜控制模块架构图

9.4　中高端磁盘阵列整体架构简析

图 9.25 所示为 NetApp 的 FAS6000 系列实物图，其中间部分为两个机头，其余均为扩展柜。

中高端盘阵一般都会配有两个控制器，不但可以做为冗余，而且可以分担后端不同的环路。图 9.26 是一个典型的双控制器，且前后端均为 FC 架构的磁盘阵列拓扑示意图。



图9.25　NetApp 公司 FAS6070 磁盘阵列

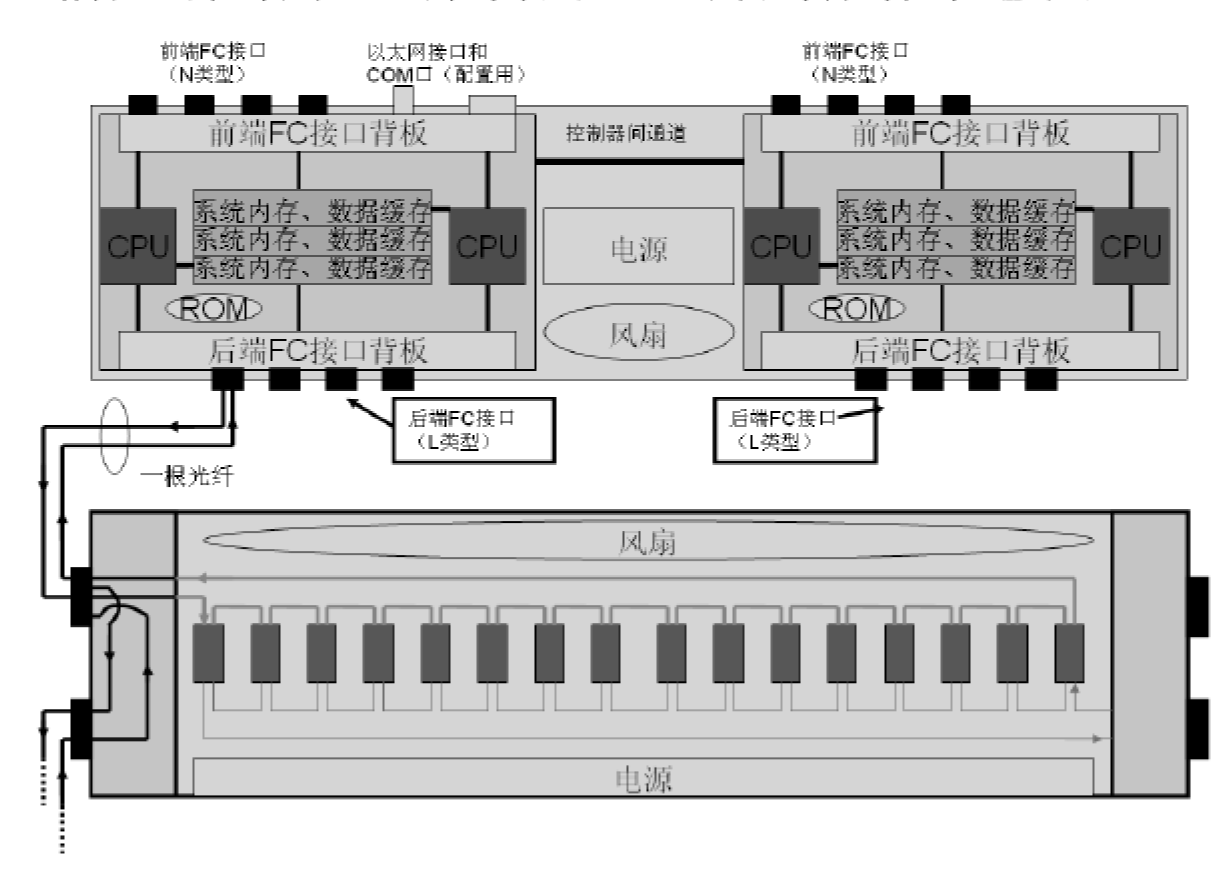


图9.26　全 FC 架构磁盘阵列

图 9.26 中所示的机头(控制器所在的机柜)中不包含磁盘。实际产品中，有些由于控制器主板比较小，机头本身也可以放入若干磁盘。但是有些高端产品的控制器主板做得比较大，IO 适配器比较多，再加上电源模块和风扇模块，造成机头内部空间不足以放下多余的磁盘。

如图 9.26 所示，每个控制器上有 4 个前端 FC 接口和 4 个后端 FC 接口。每个后端 FC 接口可以连接到一个 FC-AL 环路。为了冗余，两个控制器的后端 FC 端口必须连接到相同的扩展柜上，所以这台盘阵可以连接的 FC-AL 的 Loop 总容量为 4 个。

其实，磁盘阵列控制器本身就是一个现代计算机系统，它由 IO 设备、存储器、运算器、软件组成。

- IO 设备：包括后端 FC 适配器、前端 FC 适配器、管理用 COM 口、以太网口、LCD 液晶显示板、指示灯以及各种适配卡。控制器从后端的磁盘上提取数据，经过虚拟化之后，发送给前端的主机。控制器从前端的 FC 端口处接收主机发送的指令和数据，经过去虚拟化运算之后，通过后端 FC 端口写入扩展柜中的磁盘。这就是控制器工作的基本原理。
- 运算器：就是完成上述虚拟化和去虚拟化过程所需要的运算单元。控制器可以选用通用 CPU 来作为运算器，也可以选用或辅以专用 ASIC 芯片来完成运算。随着现代盘阵系统虚拟化功能的日益强大，软件逻辑越发复杂起来，通用 CPU 加软件就成了普遍使用的组合。至于 ASIC 等硬件只是作为一种 IO 设备而存在，辅助 CPU 进行专用逻辑的运算，并把结果返回给 CPU，目的是将 CPU 从这种专用运算中解脱出来。

- 存储器：包括高速缓存存储器和外部低速永久存储器。现代中高端盘阵几乎都是使用带 ECC 错误纠正的 DDR SDRAM 作为高速缓存。这个高速缓存既充当虚拟化软件运行时的空间，又充当两端数据流的缓存空间。
- 软件：由于虚拟化引擎越来越强大，新的功能和概念层出不穷，单纯使用精简的内核和精简的代码已经远远不能满足功能需求和开发难易度需求。所以目前很多磁盘阵列控制器都是基于某种操作系统内核的，比如 Linux、VxWorks、Windows、UNIX 等。操作系统不但提供了硬件管理层，还提供了方便的 API。这些层次的划分使开发人员只需要专心设计上层虚拟化程序而不是全盘兼顾。盘阵的操作系统和应用程序可以被存放在后端的磁盘上，也可以用专门的外部存储设备存放。NetApp 的 FAS 系列产品就是用一块 Flash 卡来存放其操作系统和应用程序的。

图 9.26 的下方是一个扩展柜，其中插了 16 块 FC 接口的磁盘。左边控制器的第一个 FC 接口通过光纤连接到了扩展柜左边的接口(一般为 SFP 接口)，线路将扩展柜内所有 16 块磁盘串接起来形成了一个 Loop。然而，16 块磁盘是远远不够的，怎么在这个 Loop 中接入更多磁盘呢？肯定是要增加扩展柜的数量，还必须将多个扩展柜中的磁盘都串接到一个 Loop 上。所以在每个扩展柜的左边接口板上都有两个 SFP 光纤接口，一个进，一个出。这样就可以把多个扩展柜中的磁盘都串在一起形成一个 Loop，接入控制器的一个后端 FC 接口。

扩展柜右边的接口板与左边构造和连接方法相同，只不过右边的接口需要连接到机头右边的控制器上，形成冗余。这样一旦左边的控制器故障，右边控制器可以立即接管所有工作。

> **提示** 扩展柜虽然说通常是一个 JBOD，但是随着 SBOD 技术的普及，扩展柜也变得复杂起来。SBOD 技术需要一系列智能芯片。柜子中的磁盘首先要插到一个背板上，背板上提供了一系列的 SCA2 型母槽。在背板上一定有某种接口来连接这一系列的芯片。通常都是将这些功能芯片单独做到一个模块上，然后将这个模块与背板对接，进而与磁盘接口对接通信。这样，如果为了实现更多的功能，可以只通过更换模块来升级，而不用大动干戈更换整个背板。所以目前几乎所有厂家的盘阵扩展柜都采用这种设计，在柜子后面可以看到两个互为冗余的模块，它们都连接到了同一个磁盘背板上。

9.4.1 IBM DS4800 控制器架构简析

在图 9.27 中可以看到两个供电模块、两个控制器模块和一个背板模块，两个控制器都连接到了背板上。

图 9.28 是 IBM DS4800 磁盘阵列的双控制器机头的背面接口图。

图中的 Drive-side connections 代表这些 FC 端口是用来连接磁盘扩展柜的，Host-side connections 代表这些 FC 端口是用作前端主机的。可以看到上下两个控制器是互为冗余的，它们同时插在机头的背板上，之间有专门的链路进行通信以交互各自的信息。

图 9.29 是用这台机头挂接 16 个扩展柜，并且全冗余的架构图。

我们可以看到，每个控制器的后端接口都连接了 4 个扩展柜，这 4 个扩展柜中的磁盘

同时位于主控制器的一个 Loop 和备用控制的一个 Loop 上。

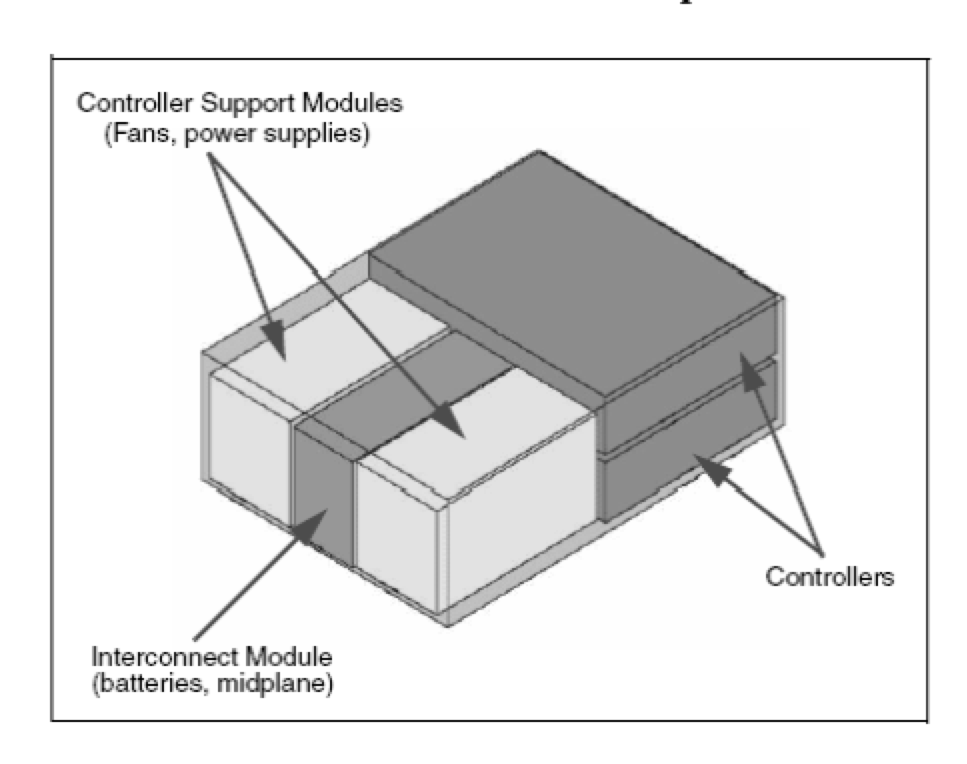


图9.27　DS4800 机头三维示意图

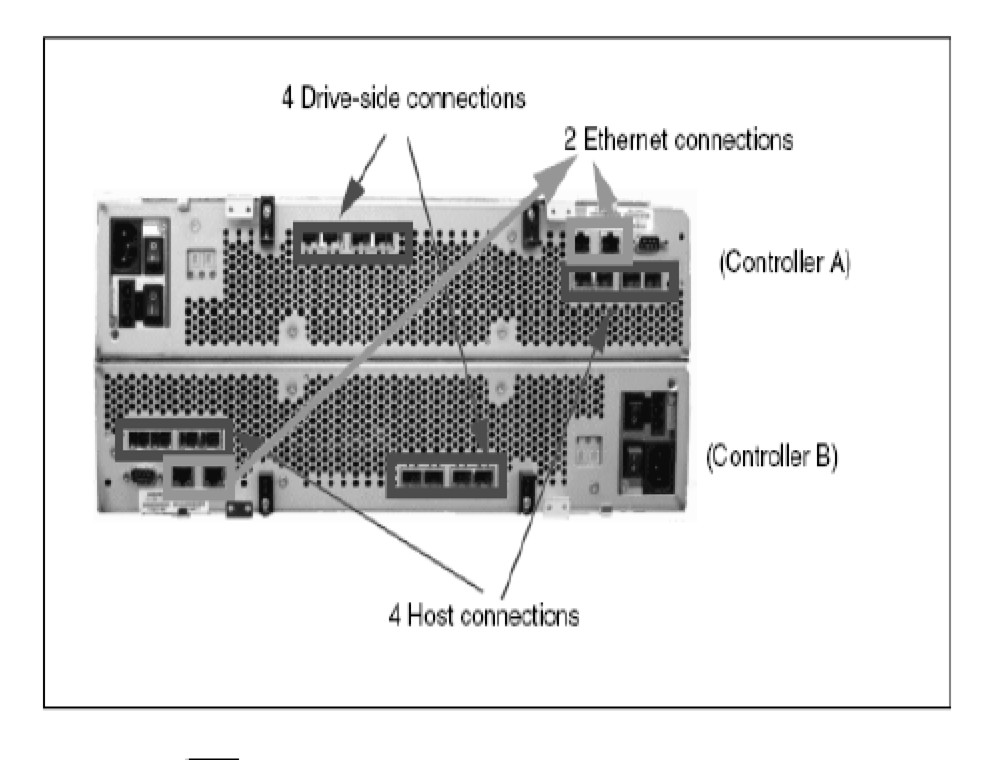


图9.28　DS4800 机头后视图

一旦机头上的主控制器发生故障，备份控制器可以立即接管所有工作，继续执行 IO 请求。因为备份控制器与主控制器一样与所有扩展柜都有连接。

同样，一旦某个扩展柜发生故障，比如电源故障，整个环路从一方来看，是被断开的。但是，其他扩展柜依然可以被访问到，办法就是通过机头上的备用控制器从尾部访问被故障扩展柜隔断的底下的扩展柜，同时主控制器从头部访问上面的控制器。

图 9.30 所示为每个扩展柜组中都有一个扩展柜故障。但是剩余的扩展柜依然可以继续使用，方法就是让主控制器和备用控制器同时工作。

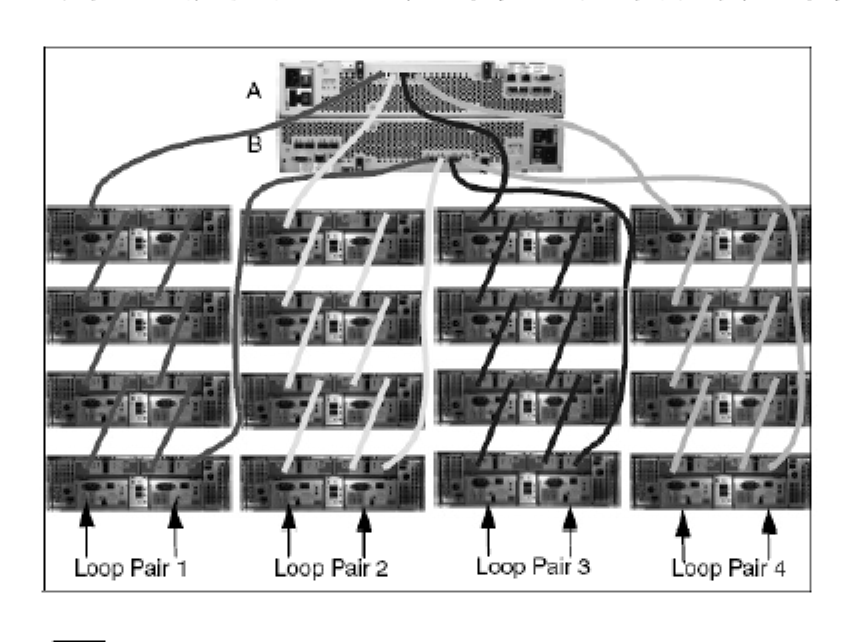


图9.29　DS4500 连接 16 个扩展柜

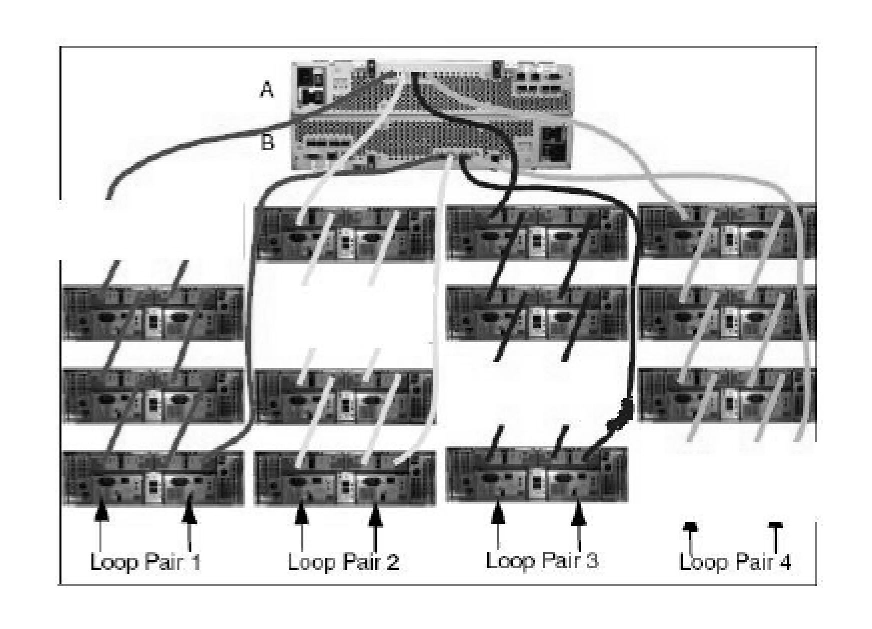


图9.30　扩展柜整柜故障时的拓扑

可以发现，如果某个 Raid Group 的磁盘全部在一个扩展柜中，那么一旦这个扩展柜故障，这个 Raid Group 将不可用。所以控制器为了获得高度可靠性，一般会尽量跨扩展柜做 Raid Group，即一个 Raid Group 中的所有磁盘各属于不同的扩展柜。这样，即使一个扩展柜失效，那么对于一个 Raid Group 来说，只是失去一块磁盘而不是全部失效，Raid Group 还可以继续工作。目前几乎所有中高端盘阵都提供这种支持。

图 9.31 是 DS4800 盘阵控制器内部简单架构图。

从图中可以看到如下部件：FC Chip、Loop Switch、Channel 及 Interconnect Module。FC Chip 是处理 FC 协议逻辑的主要芯片，全部的 FC 逻辑都在此芯片内实现。Loop Switch 在这里可以推断出其就是上文所述的 PBC 一类的芯片。其后端连接两个 FC 接口，前端分别连接位于两个控制器上的两块 FC Chip，这样做的目的是为了充分冗余。Channel 的意思就是指连接到一个 Loop Switch 上的两个 FC 端口，两个端口组成一个 Channel，没有实际物理意义。Interconnect Module 其实是一个背板，用于控制器之间的通信。

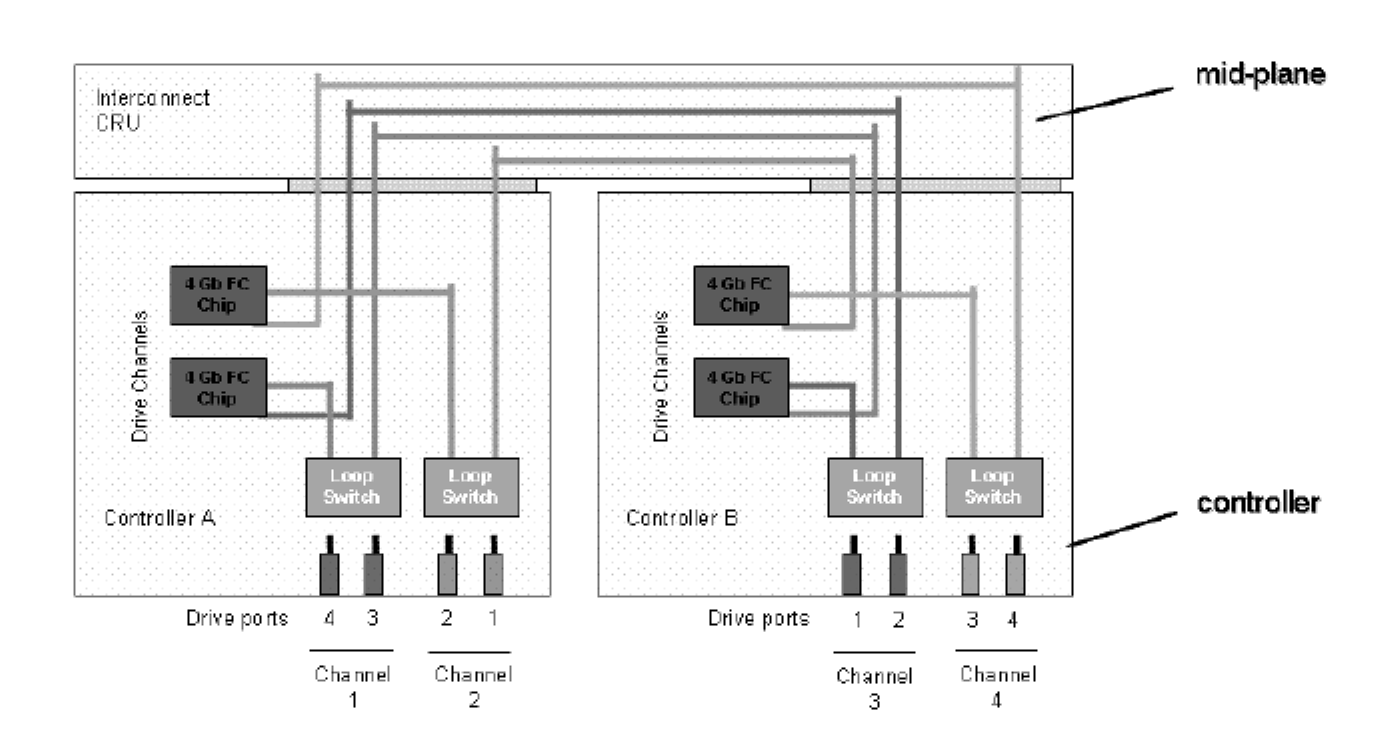


图9.31 DS4800 机头控制器简单架构图

提示

推论：假设某一时刻，左边控制器下方那个 FC Chip 失效，则控制器 A 的 4 号 FC 接口与这个 Chip 的通路便会断掉。此时只能通过控制器 B 的 1 号 FC 接口访问最左边的 4 个扩展柜。查看一下连线，可以发现控制器 B 的 1 号 FC 口其实是通过 Interconnect Module 连接到了控制器 A 的上面的 FC Chip。

所以，最终数据是通过控制器 B 右边的 Loop Switch 流向 Interconnec Module，然后流到控制器 A 上面的 FC Chip。也就是说，最终达到了同一个控制器内部两个 FC Chip 之间的冗余备份。而此时对前端主机来说并没有影响，主机还是连接控制器 A 来读写数据。同样，如果控制器 A 上的某个 Loop Switch 失效，则数据全部通过控制器 B 对应的 Loop Switch 流向 Interconnect Module，最后还是流回到控制器 A，对主机并没有影响。

但是，如果同一个控制器上的两个 FC Chip 或两个 Loop Switch 都失效了，甚至整个控制器故障了，那么所有 IO 访问就都要转移到另外的一个控制器上。此时，对主机端就会产生影响，需要多路径软件和盘阵端配合参与故障切换的动作。

实际上，DS4800 的控制器不一定是上述的切换模式。但是在理论上，所有产品的设计都应该尽量不切换控制器，因为切换控制器会对主机端造成影响。

图 9.32 为 DS4800 扩展柜连接示意图。

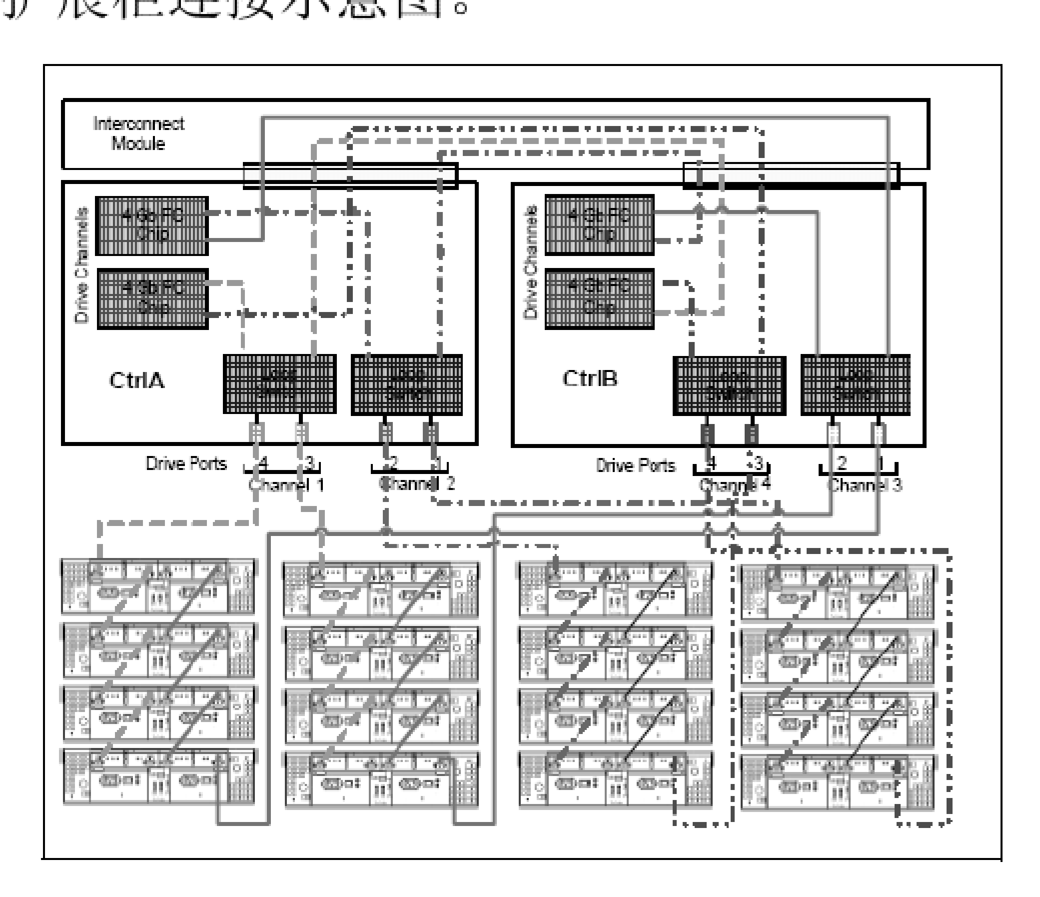


图9.32 DS4800 扩展柜连接通路示意图

图 9.33 是 DS4800 一个控制器的内部详细架构图。

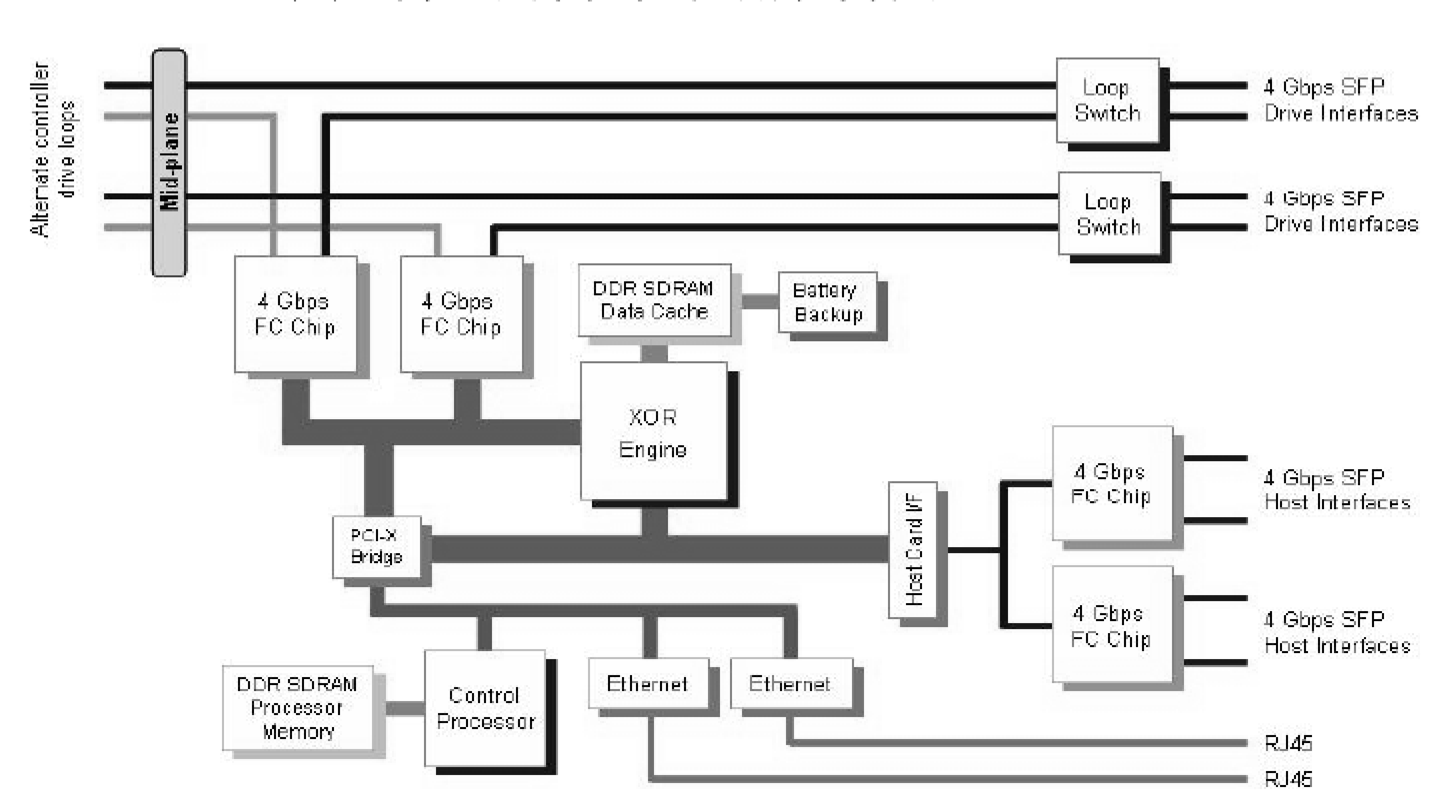


图9.33　DS4800 控制器详细架构图

- 2.4 GHz Xeon processor 运行 Vxworks 实时操作系统
- 每个控制器 2、4、8GB 数据缓存 RAM——数据专用
- 每个控制器 512MB RAM 系统缓存——操作系统运行内存
- XOR ASIC Engine 专用芯片，硬件 RAID 运算引擎
- 64bit/133MHz——1GB/s 带宽的 PCIX 总线

9.4.2 NetApp FAS 系列磁盘阵列控制器简析

1. FAS2050 磁盘阵列

图 9.34 为 FAS2050 磁盘阵列控制器的后视图。

从图 9.34 中可以看到上下两个控制器，每个控制器各有两个 FC 接口。

2. FAS3050 磁盘阵列

图 9.35 为 FAS3050 磁盘阵列控制器后视图。

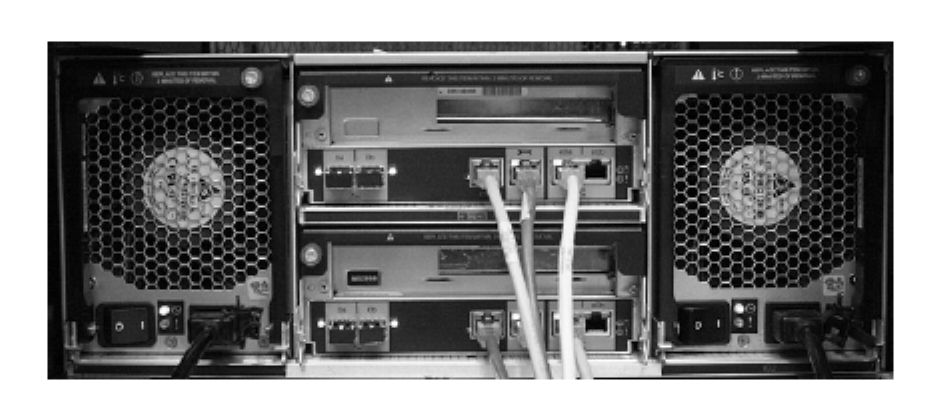


图9.34　FAS2050 控制器实物后视图

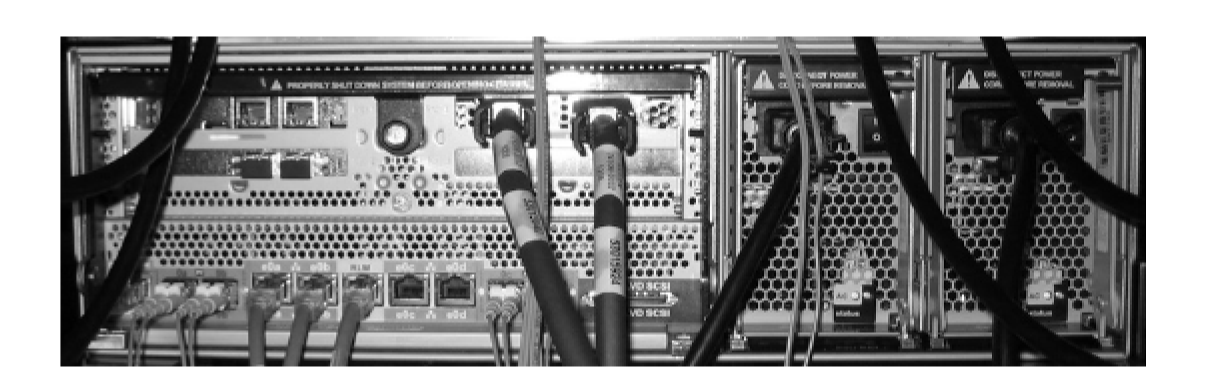


图9.35　FAS3050 控制器实物后视图

图 9.35 为单个 FAS3050 的控制器机头后视图。图中只是一个控制器，如果要达到完全冗余，可以用两台控制器形成 Cluster 结构。FAS3050 有 4 个板载 FC 接口，可以通过插 PCIX 接口的扩展卡来扩充 FC 接口的数目。每个控制器提供 4 个扩展卡槽位，可以插接 FC 卡、以太网卡、TOE 卡和 ISCSI 卡等扩展卡。

NetApp 的 FAS 产品，从 FAS3000 系列开始，由于处理功能增强，扩展槽位增多，所以一个控制器就占满一个机头的空间。两台控制器之间需要通过 NVRAM 卡上的 Infiniband 网络来形成 Cluster。

3. FAS6070 磁盘阵列

图 9.36 为 FAS6070 磁盘阵列控制器后视图。

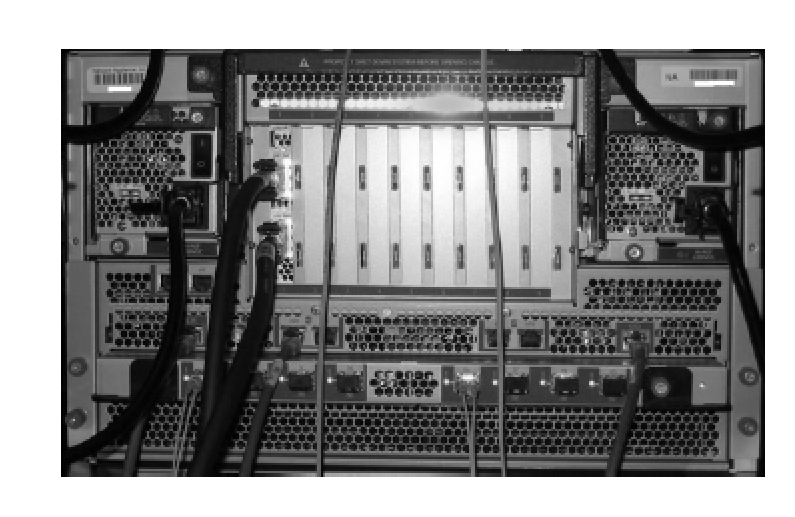


图 9.36 FAS6070 单个控制器实物后视图

FAS6070 是 NetApp 公司比较高端的设备，其内部后端总线的总理论带宽可达 32GB/s。有 9 个扩展槽位，可以插接扩展 FC 适配器以便连接更多的扩展柜，或 TOE 卡、以太网卡等其他适配器。

4. DS14MK2FC 磁盘扩展柜

图 9.37 所示为 DS14MK2FC 磁盘扩展柜的前视图。

图 9.38 所示为 DS14MK2FC 磁盘扩展柜的后视图。

图 9.37 和图 9.38 所示的是用于连接 FAS 系列控制器的磁盘扩展柜，每个柜子可以插 14 块 FC 接口的磁盘。从后视图中可以看出其与 FAS2050 控制器的拓扑比较像，不要搞混。前文说过，现代磁盘扩展柜都是用双模块设计，模块中有半交换 SBOD 芯片。盘柜中所有的磁盘利用其 SCA2 接口连接到背板上，这个物理接口中所包含的两个逻辑接口各通过电路连接到一个扩展柜控制模块上，形成双 Loop 冗余。



图 9.37 DS14MK2FC 磁盘扩展柜前视图

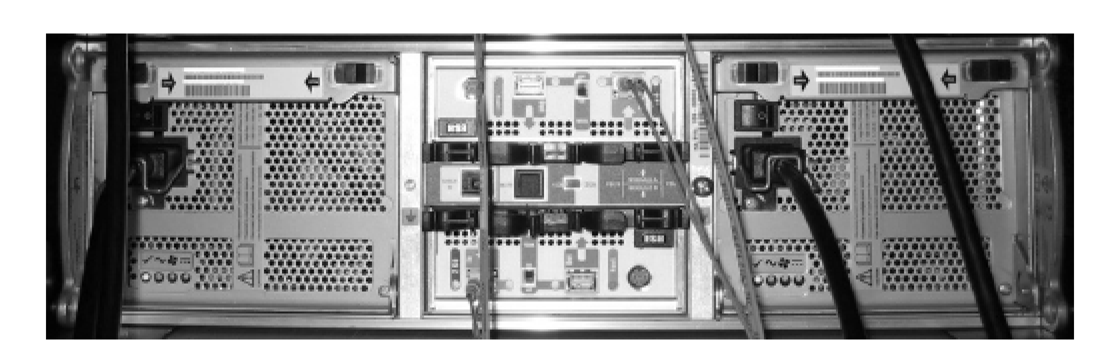


图 9.38 DS14MK2FC 磁盘扩展柜后视图

9.4.3 IBM DS8000 简介

DS8000 系列利用两台 P 系列主机充当控制器。上文中说过，盘阵控制器架构本质上与主机架构无异，DS8000 就是这样一个例证。图 9.39 和图 9.40 为 DS8000 实物图。

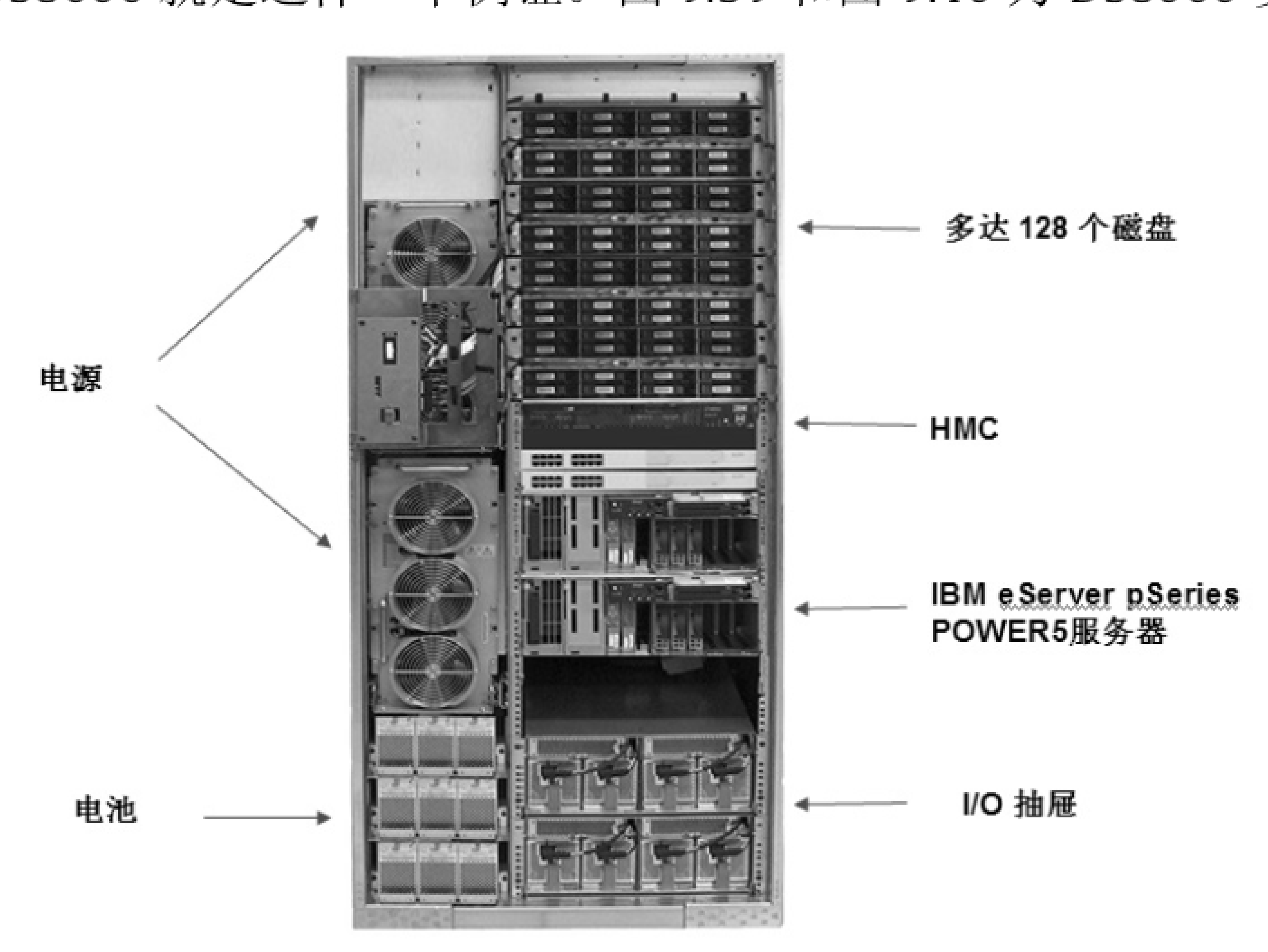


图 9.39 DS8000 主机架

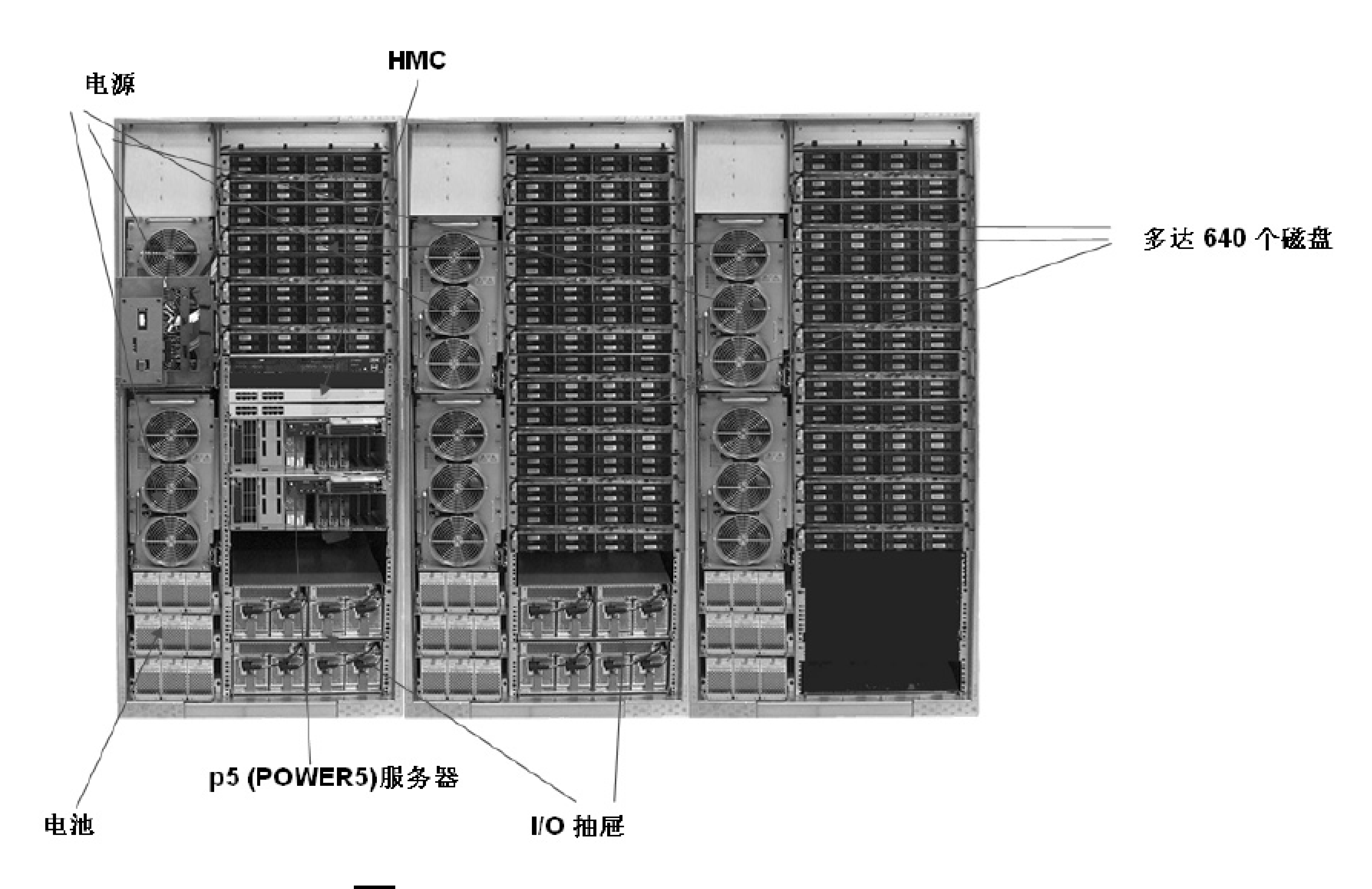


图9.40　DS8000 盘阵主机架和扩展机架

IBM DS8000 磁盘阵列是 IBM 磁盘阵列产品线中的最高端产品。它利用两台 IBM 的 P 系列服务器作为控制器运行 AIX 操作系统，操作系统之上运行了 DS8000 的存储虚拟化引擎和管理软件。

9.4.4　富士通 ETERNUS6000 磁盘阵列控制器结构简析

图 9.41 为富士通 ETERNUS6000 机柜布局示意图。

与 NetApp FAS6070 和 IBM DS8000 系列一样，高端磁盘阵列系统由于需要提供极高的存储容量，只连接几个磁盘扩展柜已经无法满足要求。所以就需要将更多的磁盘扩展柜装入扩展机架中，然后统一联入控制器或者串联到其他机架。

图 9.42 为 ETERNUS6000 控制器架构示意图。

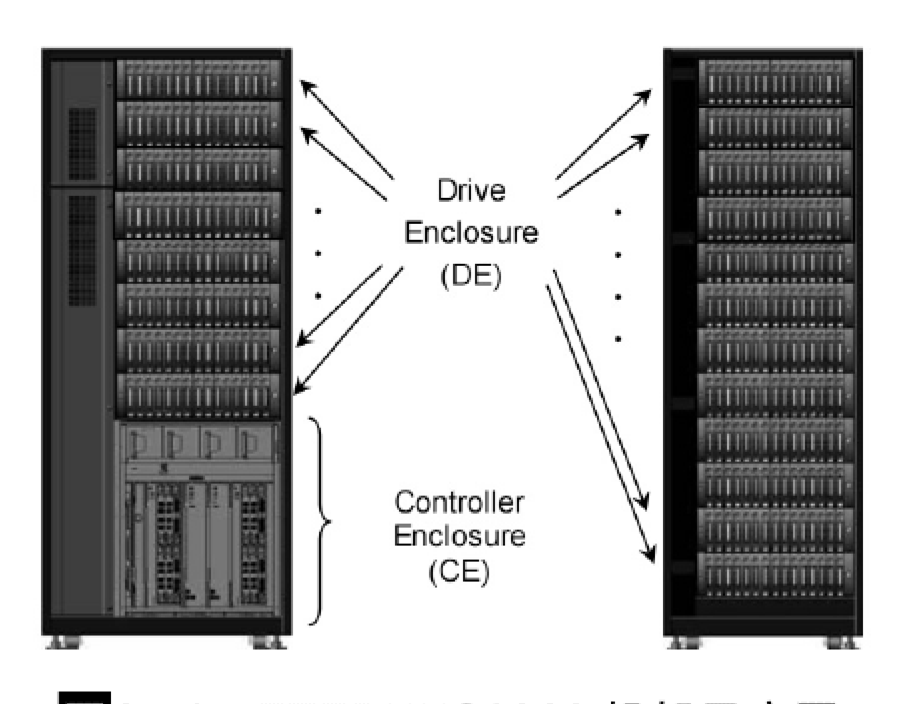


图9.41　ETERNUS6000 机架示意图

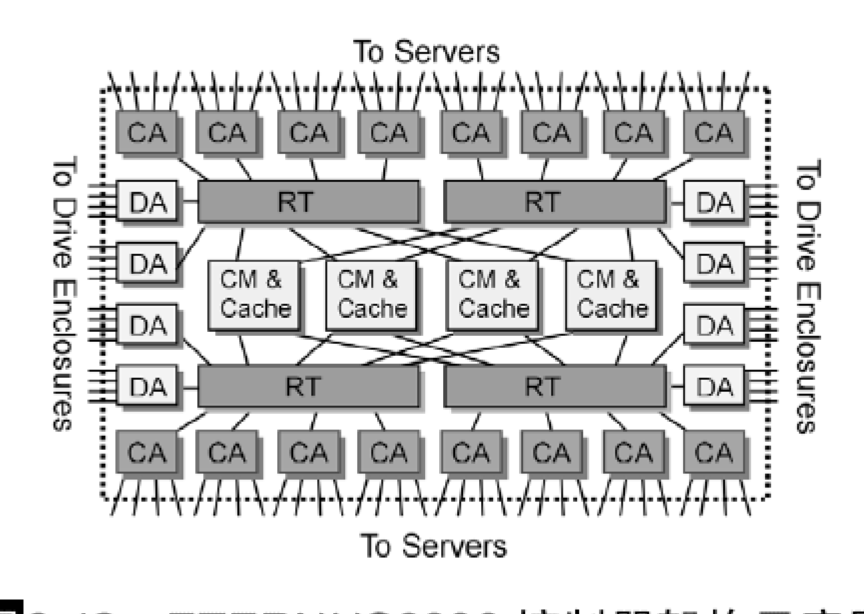


图9.42　ETERNUS6000 控制器架构示意图

ETERNUS6000 整体架构简单拓扑如图 9.42 所示。

- CM(Control Module)为控制器模块。CM 是整个磁盘阵列的计算中心，每个 CM 包含一个 2.8GHz 的 CPU 和最大 16GB 的 RAM。RAM 既作为盘阵本身软件的运行内存，又作为盘阵数据缓存(图中所示的 Cache)。整个盘阵系统最多可以安装 4 个 CM，由于每个控制器柜只包含一个 CM，所以整个系统最多可以连接 4 个 Controller

Enclosure。

- RT(Router)为路由器。RT 模块相当于普通服务器架构中的南桥控制器(IO 控制器)。不同的是这里的 RT 是一个全局桥，它可以桥接整个系统中的所有 CPU、RAM 和外设(FC 接口卡)，使所有部件之间实现高速通信。每个系统最大可以接入 4 个 RT。
- DA(Driver Adapter)为磁盘适配板。每个适配板上接有 4 个 FC-AL 接口，用于接入 FC-AL 的 Loop。DA 实际上就是盘阵的后端接入点。每个 RT 可以接入 2 个 DA，所以每个系统最多接入 8 个 DA，从而可以挂接 32 条 Loop。但为了冗余，每个 Loop 需要同时连接 2 个 DA，所以实际可用 Loop 减半，为 16 Loops。
- CA(Channel Adapter)为通道适配板。CA 实际上就是前端 FC 接口适配板，每个板子上包含 4 个 FC N 类型端口，用于接入 FC 交换设备或者直接连接主机服务器的 FC 适配卡。每个 RT 最多接入 4 个 CA，所以整个系统最多可以接入 16 个 CA，最多提供 64 个 FC 协议 N 类型前端接口。

1. SBOD

扩展柜同样采用了双模块板设计，可以保证两条路径到达同一块磁盘。图 9.43 中的“骨干交换机”其实就是指 SBOD 所采用的半交换式芯片。

2. 循环镜像的写缓存

图 9.44 为 ETERNUS6000 控制器间循环缓存镜像写示意图。

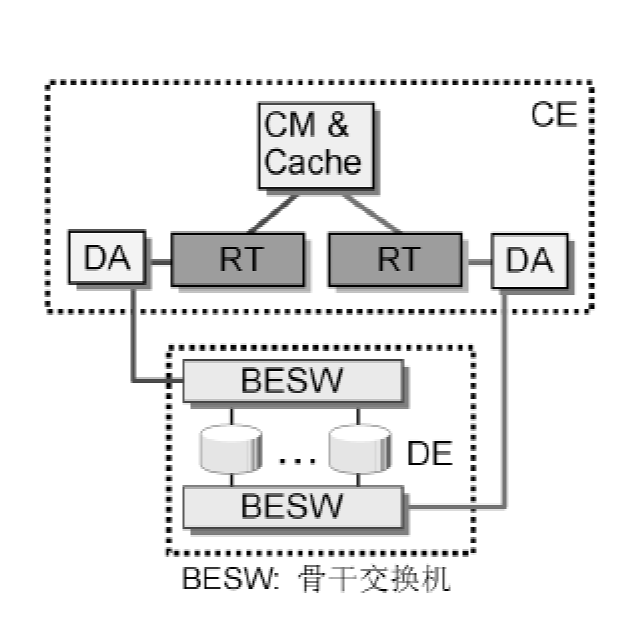


图9.43 ETERNUS6000 扩展柜架构

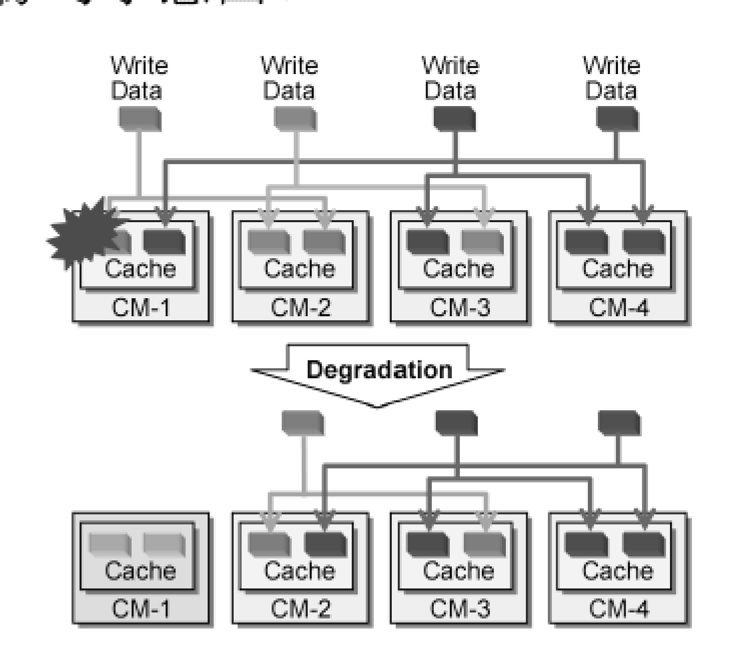


图9.44 ETERNUS6000 循环缓存镜像写示意图

数据在被写入任何一个控制器的 Cache 时，系统会将写入的数据复制到其他控制器的缓存中做冗余。一旦在数据还没有写入硬盘之前，某个控制器发生了故障，写缓存镜像技术可以保证数据不会丢失，可以将数据从镜像缓存中写入硬盘。读缓存不需要这种技术，因为读操作只是将数据从硬盘上读入 RAM，如果此时控制器故障，磁盘中的数据依然存在。最重要的是缓存镜像技术会浪费宝贵的缓存容量，读操作没有必要实现缓存镜像。

3. 跨扩展柜做 Raid Group

图 9.45 所示为 ETERNUS6000 跨扩展柜的 Raid Group 示意图。

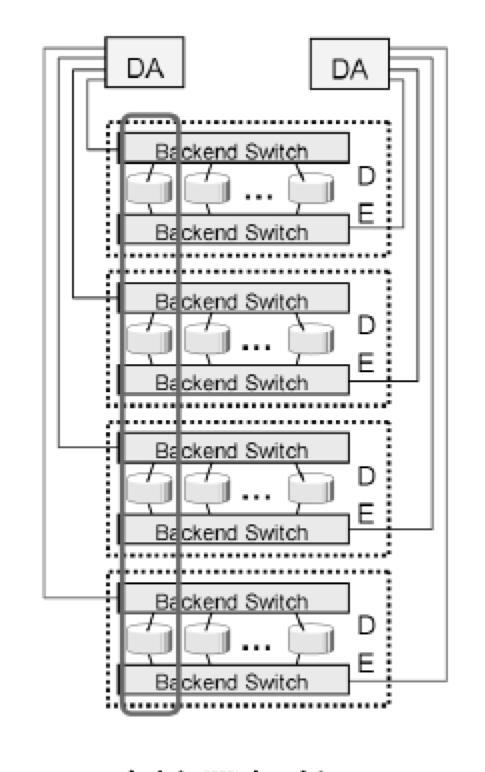


图9.45 跨扩展柜的 Raid Group

前面也介绍过，为了获得足够的冗余性，很多厂家的盘阵产品一般都选择将不同扩展柜中的硬盘容纳到一个 Raid Group 中，而不是让一个 Raid Group 包含同一个扩展柜中所有的磁盘。

9.4.5 EMC 公司 CX 及 DMX 系列盘阵介绍

1. CX 系列产品

CX 系列产品性能如下。

- 充分冗余体系结构。
  - 双存储处理器。
  - 电源、冷却、数据路径、独立电源。
  - 无间断操作。
  - 在线软件升级。
  - 在线硬件更改。
- 高级数据完整性。
  - 镜像写缓存。
  - 发生电源故障时将写缓存转储到磁盘。
  - SNiiFFER：扇区检查实用程序。
  - 点对点 DAE 设计。
  - 具有无中断故障切换的双 I/O 通道。
- 分层容量。
  - 15K：36 GB、73 GB、146 GB；
    10K：73 GB、146 GB、300 GB。
  - 500 GB SATA Ⅱ。
- 5～480 个磁盘。
- 灵活性。
  - 混合驱动器类型。
  - 混合 RAID 级别。
  - RAID 级别 0、1、1+0、3 和 5。
- 高达 16 GB 的可调式缓存。

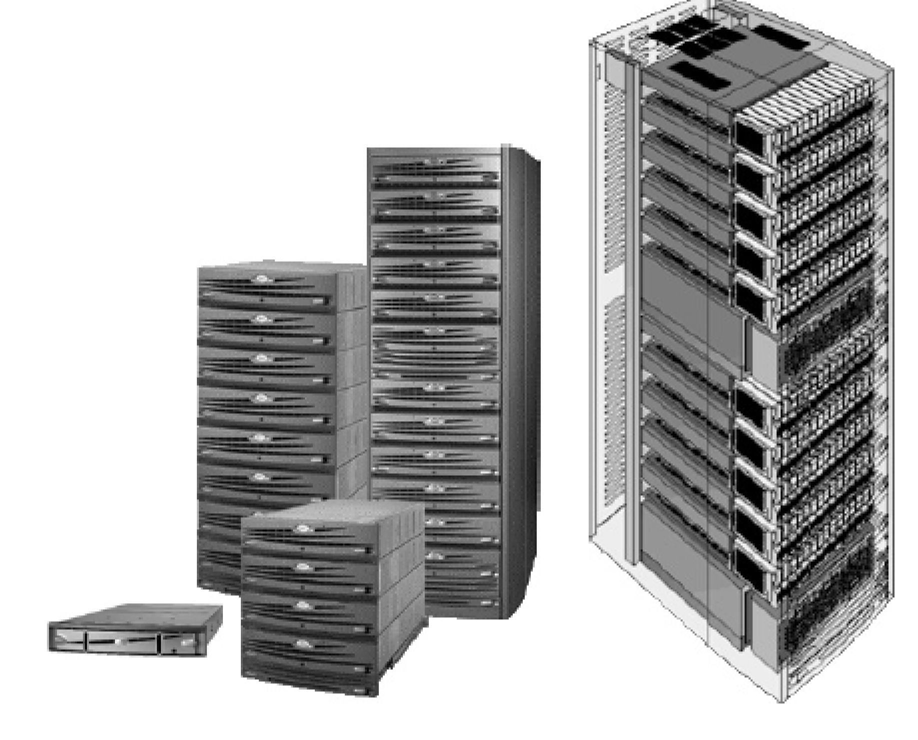


图 9.46 CX 系列产品实物图和透视图

图 9.46 为 CX 系列的实物图和透视图。

图 9.47 为 CX700 的控制器架构示意图。

图 9.48 为 CX700 控制器实物图。

2. Symmetrix DMX-3 系统概述

- 直连矩阵体系结构。
  CPU 与内存之间采用点对点直连访问。每一个控制器都有其自己到达每一目的地的专用通道。直连矩阵底板最多有 128 个全部是直连、专用而且不共享的独立通道，如图 9.49 所示。
- 每个控制器 8 个 1.3 GHz PPC 处理器。

- 最多 12 个通道控制器。
  - 8 端口 2 Gb 光纤通道。
  - 8 端口 ESCON。
  - 4 端口多协议——2 Gb FICON、iSCSI 和用于 RDF 的千兆以太网。
- 最多 8 个磁盘控制器。
  - 每个磁盘控制器对最多 480 个驱动器。
  - 支持无中断添加控制器。
- 高达 512 GB 全局内存(256 GB 可用)。

  带有内存保险存储保护的镜像 DDR 技术。

图 9.50 所示为 Symmetrix DMX-3 的实物图，左边为控制器机柜，右边为磁盘扩展柜机柜。

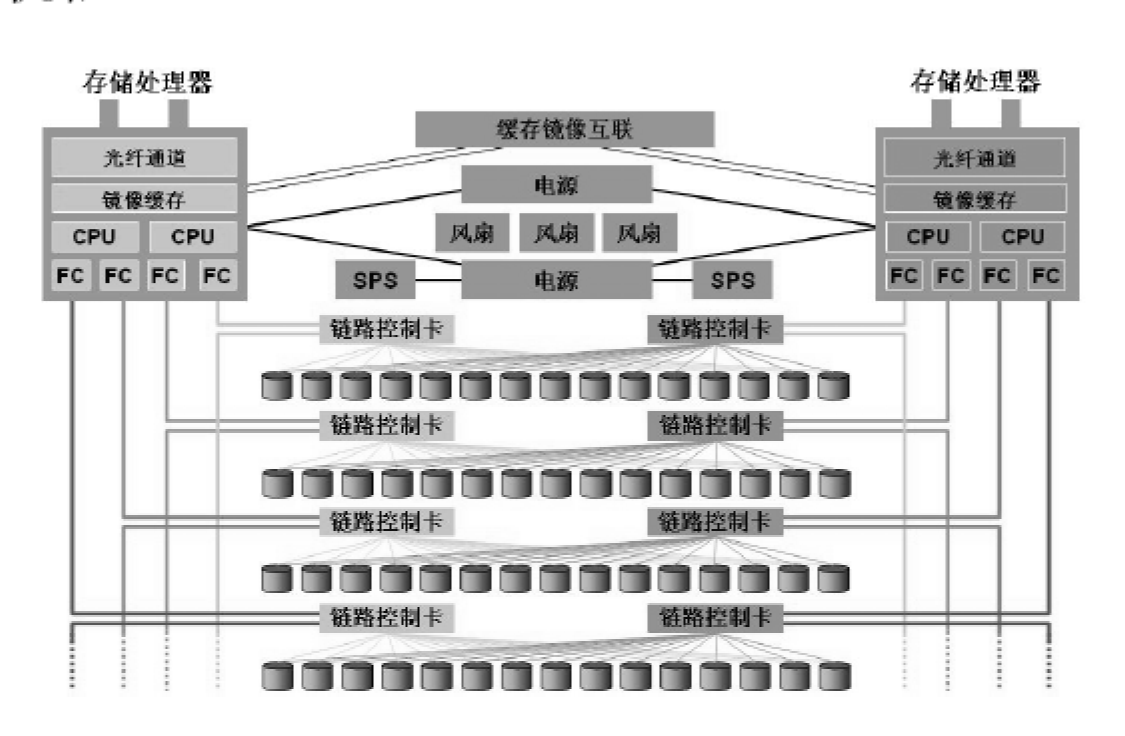


图9.47 CX700 的控制器架构示意图

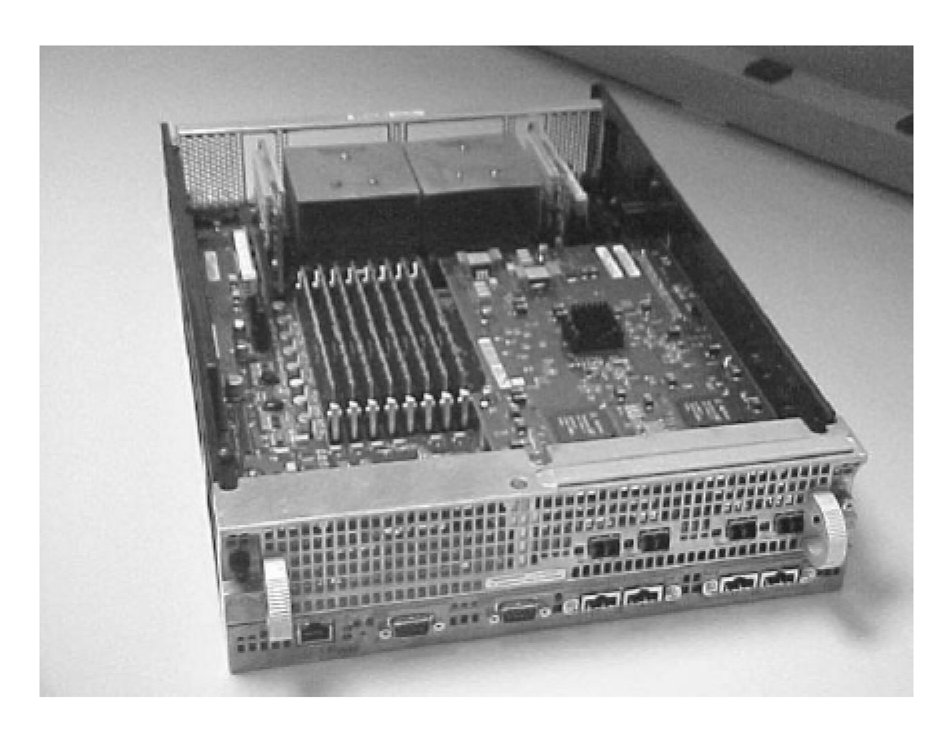


图9.48 CX700 控制器实物图

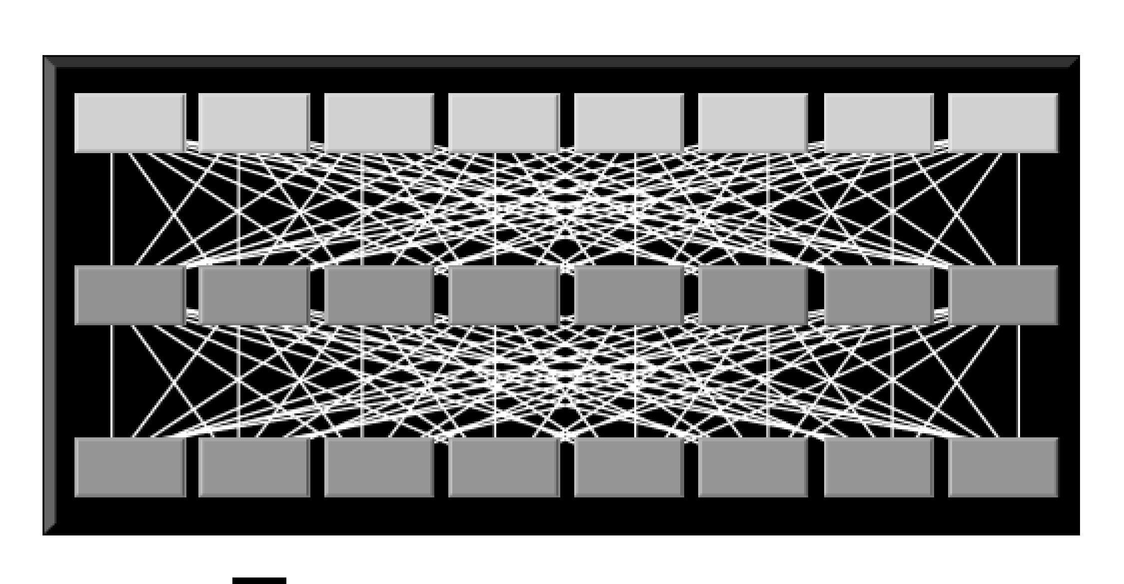


图9.49 Symmetrix 矩阵示意图

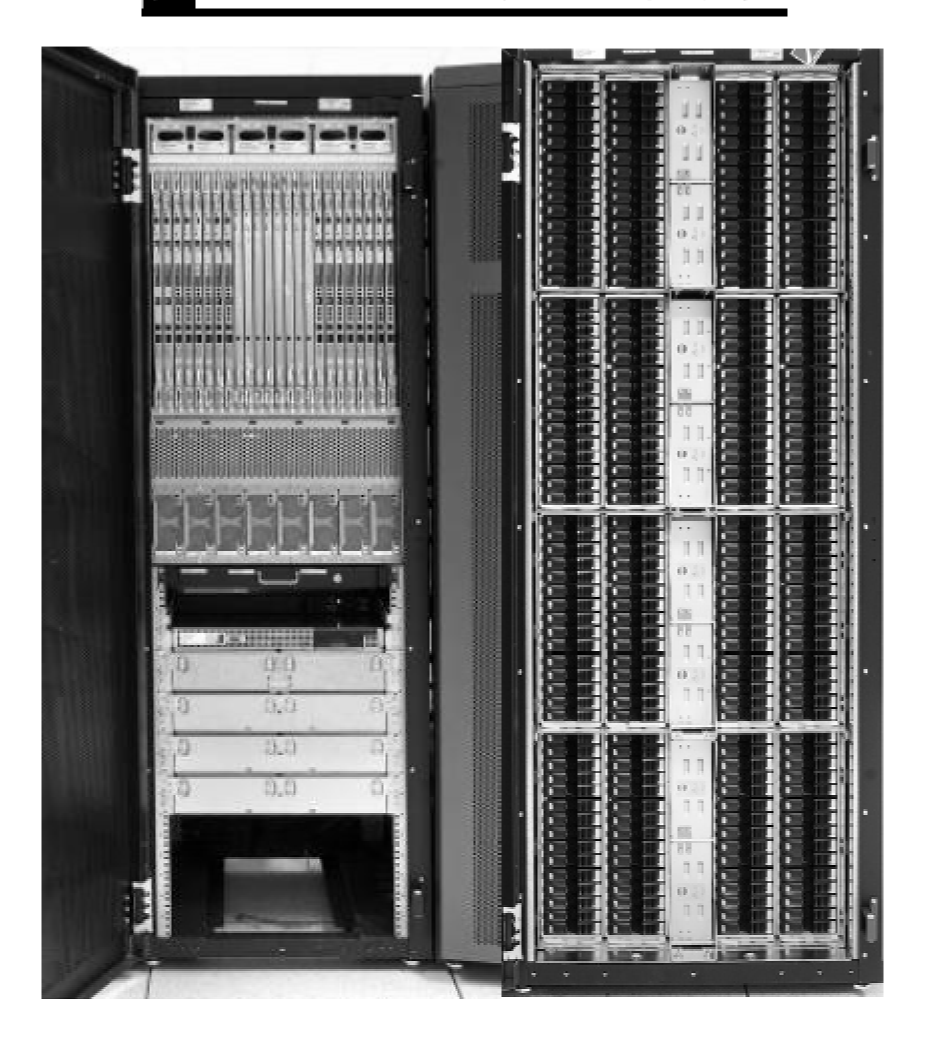


图9.50 Symmetrix DMX-3 实物图

9.4.6 HDS 公司 USP 系列盘阵介绍

USP 系列机器为目前 HDS 存储产品中最高端的机器。图 9.51 所示为 USP 系列机柜实物图。图 9.52 所示为 USP-V 系列的虚拟化功能示意图。图 9.53 所示为 USP 系列控制器架构

逻辑示意图。



图9.51 机柜实物图

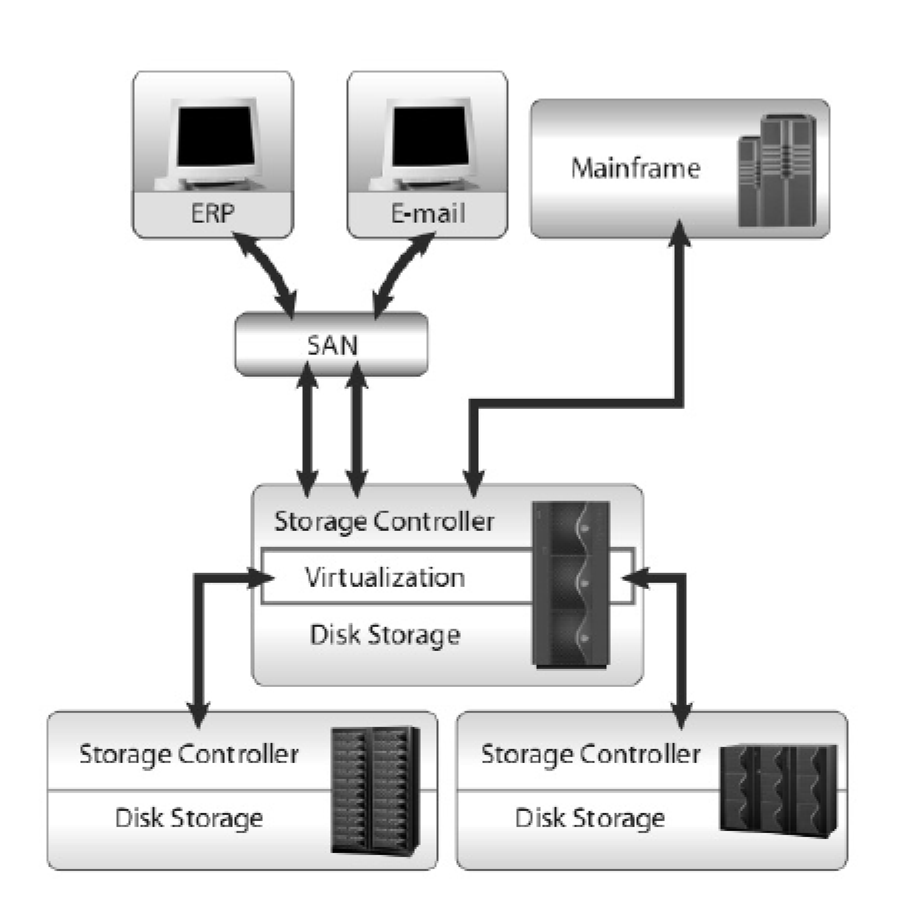


图9.52 USP-V 系列虚拟化示意图

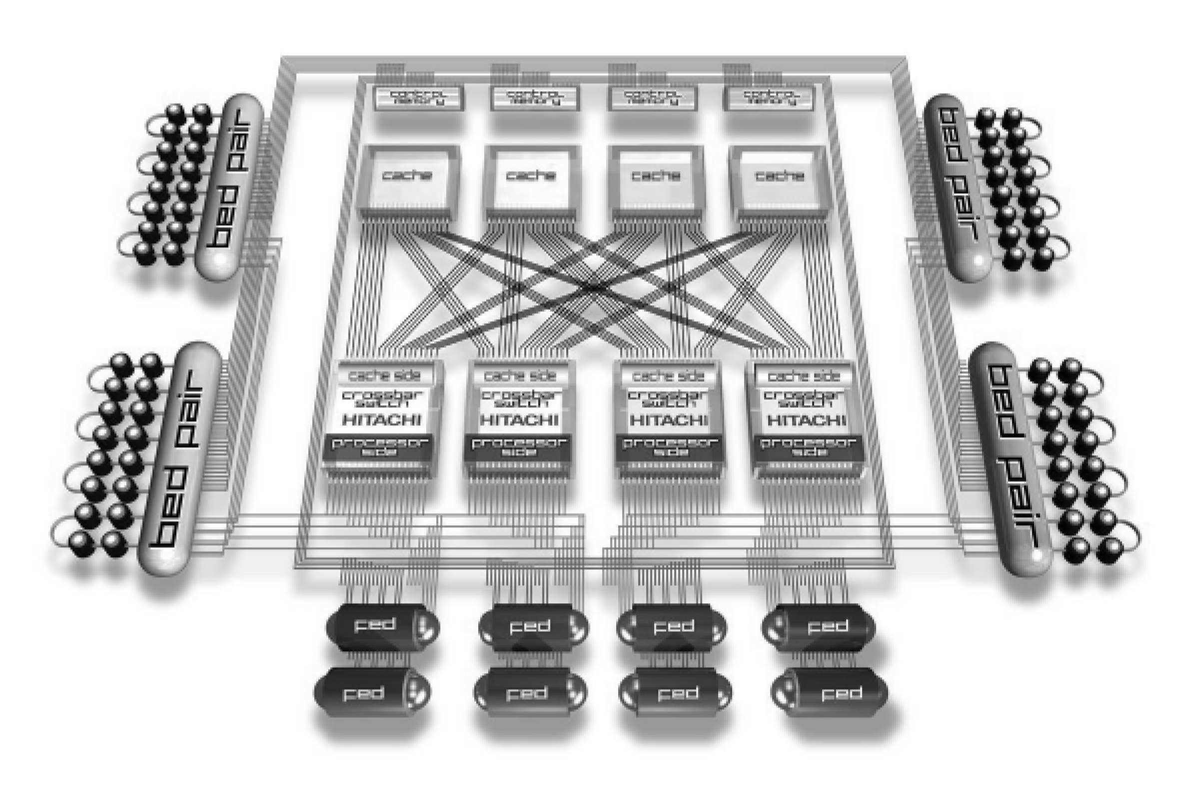


图9.53 控制器架构示意图

9.5 磁盘阵列配置实践

9.5.1 基于 IBM 的 DS4500 盘阵的配置实例

在拿到一台崭新的磁盘阵列之后，必须对其进行相关的配置，才能让其发挥功能。这些配置包括配置 LUN，各个 LUN 的参数，配置 LUN 映射。这三项配置是使一个盘阵可以用来存取数据的最基本配置。下面就以一台带有两个扩展柜的 DS4500 磁盘阵列的配置为例，向大家演示一下这些配置的基本步骤。

Storage Manager 软件是 IBM 公司开发的专门针对其 DS4000 系列盘阵的配置工具，这个工具可以运行在 Windows 操作系统上，通过以太网与磁盘阵列通信，从而实现配置。

在配置阵列参数之前，我们先来看一下这个阵列将被用于一个什么样的环境。这台阵列是一个小型公司购买的，准备用于存放公司的机密文件、SQL Server 数据库文件、静态 Web 网站文件和内部邮件数据。整体拓扑结构如图 9.54 所示。

其中 SQL Server 数据库服务器需要 3 个 LUN 存放数据，总容量 1TB＋。邮件和网站各

需要一个 LUN，大小分别为 300GB＋和 50GB＋。FTP 服务器需要 2 个 LUN，400GB＋。

所有服务器操作系统均为 Windows 2003 Enterprise Server。每台服务器上均要安装 2 块 2Gb 速率的 FC HBA 卡，而且必须安装多路径软件。

在一台普通 PC 上安装盘阵随机带的 Storage Manager 软件，然后将这台 PC 用双绞线连接到盘阵的以太网口上，将 PC 的 IP 地址配置成与盘阵初始 IP 地址相同的网段。之后，通过软件主界面添加盘阵的 IP 地址，这样软件就可以通过 TCP/IP 协议与盘阵进行通信了。

从图 9.55 中可以看到这套设备共有 3 个柜子，控制器所在的柜子是 0 号柜子，也就是图中右边显示的 Controller Enclosure，这个柜子中包含两个互为备份冗余的控制器 A 和 B。Driver Enclosure1 和 Driver Enclosure 2 是两个磁盘扩展柜，每个柜子中包含 14 块物理磁盘。所有磁盘的总容量为 2552GB，这在主界面左边栏中也有提示。

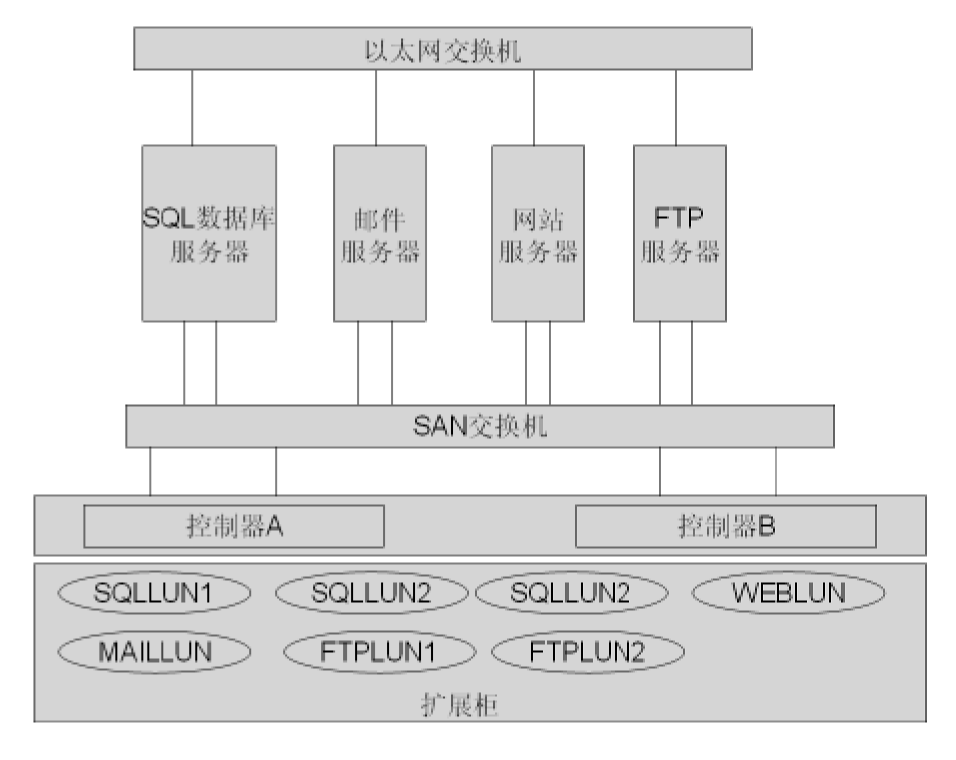


图9.54　某公司 IT 系统后端拓扑图

图9.55　Storage Manager 的主界面

单击按钮可以查看柜子的供电、散热、温度、控制器电池、插卡、SFP 模块等硬件部件信息，如图 9.56 所示。

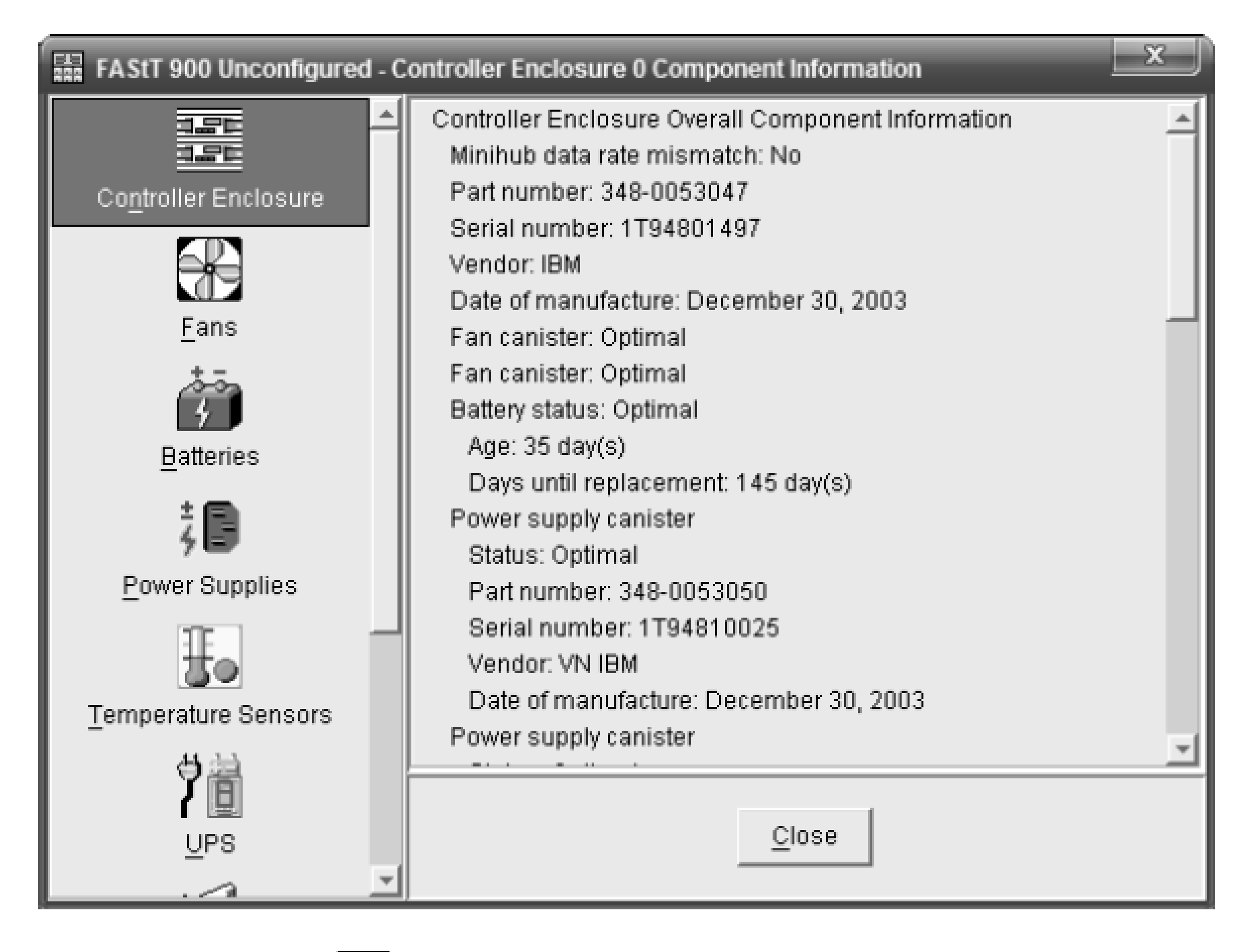


图9.56　机器环境监控窗口

1. 配置 LUN

在主界面中右击 Unconfigured Capacity，创建逻辑磁盘(即 LUN)，如图 9.57 所示。

1】　出现选择主机类型的对话框，选择这个逻辑磁盘将要为何种类型的操作系统使用，

这里我们选择 Windows 2000/Server 2003 Non-C，即“Windows 2003 非集群”类型，如图 9.58 所示。

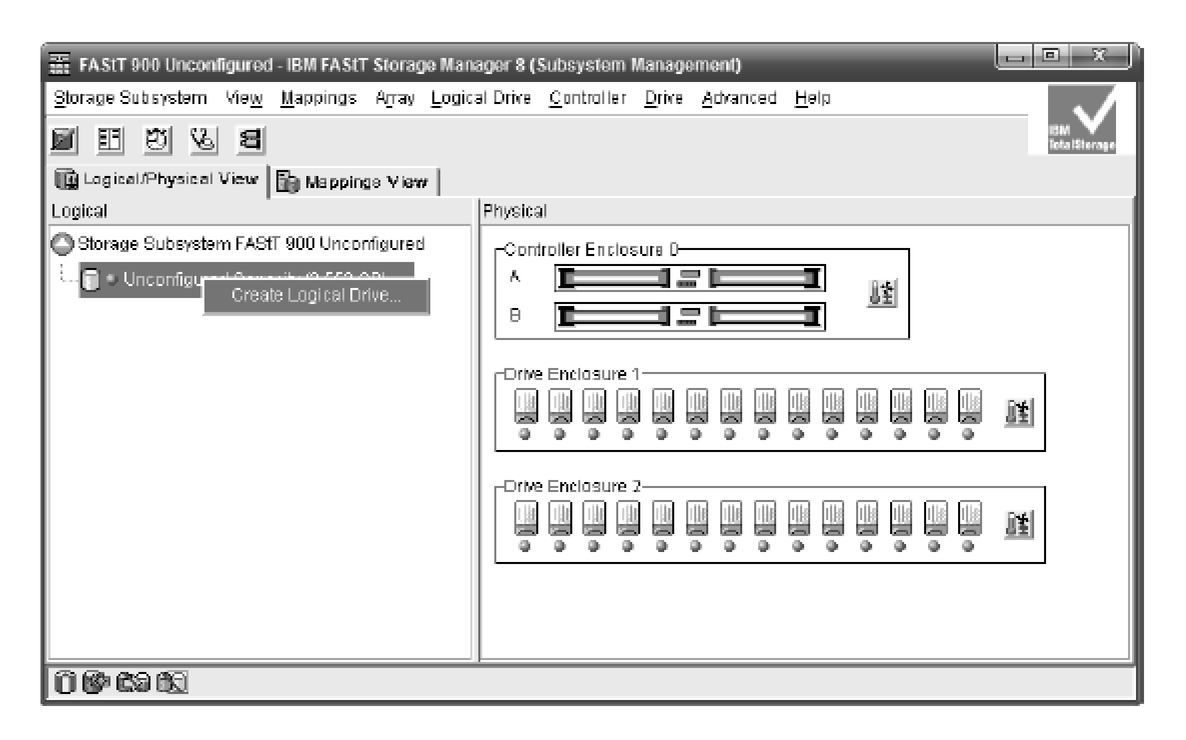


图9.57　创建 LUN

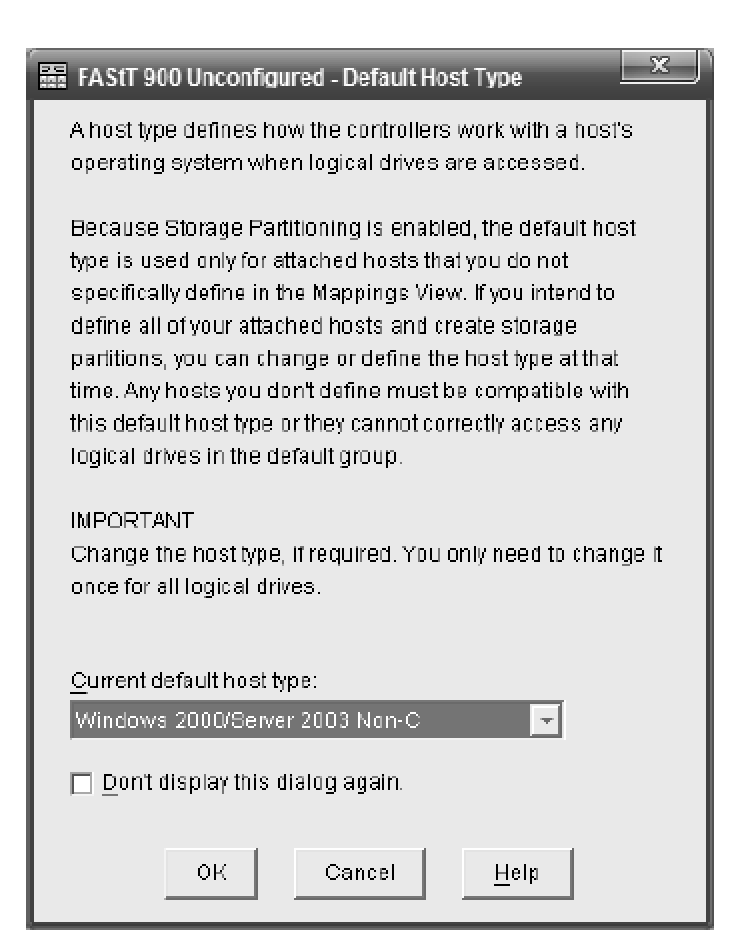


图9.58　选择 LUN 的类型

提示　有些磁盘阵列会针对不同的操作系统提供不同类型的 LUN。虽然 LUN 对于操作系统来说只是一块裸磁盘，但是每种操作系统上的文件系统在使用 LUN 的行为上是相差异的。LUN 的 0 号 LBA 为 MBR，占用了一个扇区。而文件系统一般使用比扇区更大的逻辑块来做为分配单元，所以有些文件系统对于 LBA 的编号就要从 LBA1 开始，即 MBR 的下一个 LBA。对于一些盘阵控制器来说，它们管理 LUN 可能也是用块来做为一个最小单元。但控制器却不理解 MBR，只认为文件系统会从 0 开始顺序编号。这样，就产生了块不对齐的现象，从而影响了性能，如图 9.59 所示。所以，有些以块为单元对 LUN 做管理的盘阵控制器会针对不同文件系统制作对应的 LUN。

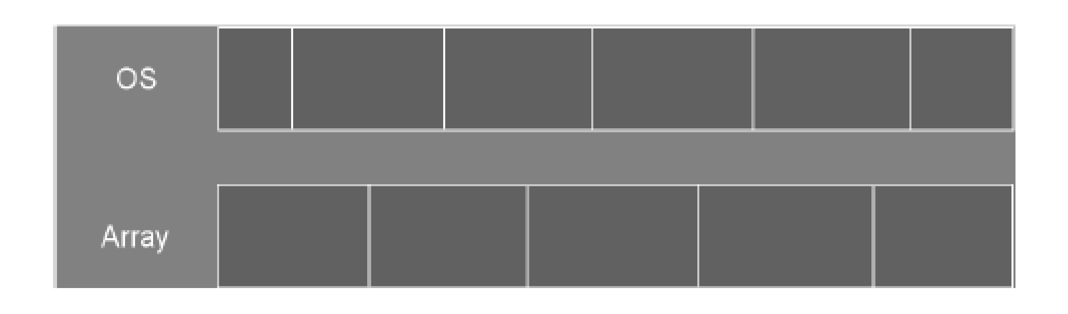


图9.59　文件系统与盘阵的 LUN 块不对齐示意图

在图 9.60 所示的对话框中，需要先创建 RAID Group(或 Arrays)，然后在建好的 RG 中再划分 LUN。数据库对于 IO 性能是要较高要求的，所以给数据库的 3 个 LUN 各分配一个 RG，充分保证数据库的 IO 性能。

2】 选择相应的 RAID 级别。如果想手动选择组成 RAID Group 的磁盘，选中 Manual 单选按钮。然后在下方磁盘列表中，按住 Ctrl 键选择组成这个 RG 的所有磁盘成员，最后单击 Apply 按钮，如图 9.60 所示。如果想让程序自动生成各种大小的 RG，则要选中 Automatic 单选按钮，如图 9.61 所示。

3】 这里让系统自动为我们计算。在 RAID 5 级别下，系统计算出的各种容量、磁盘数量的 RG 组合。Channel Protection 指的是让 RG 中所有磁盘成员分布在两个扩展

柜中，充分保证冗余性。我们选择 5 块盘的 292GB 容量。

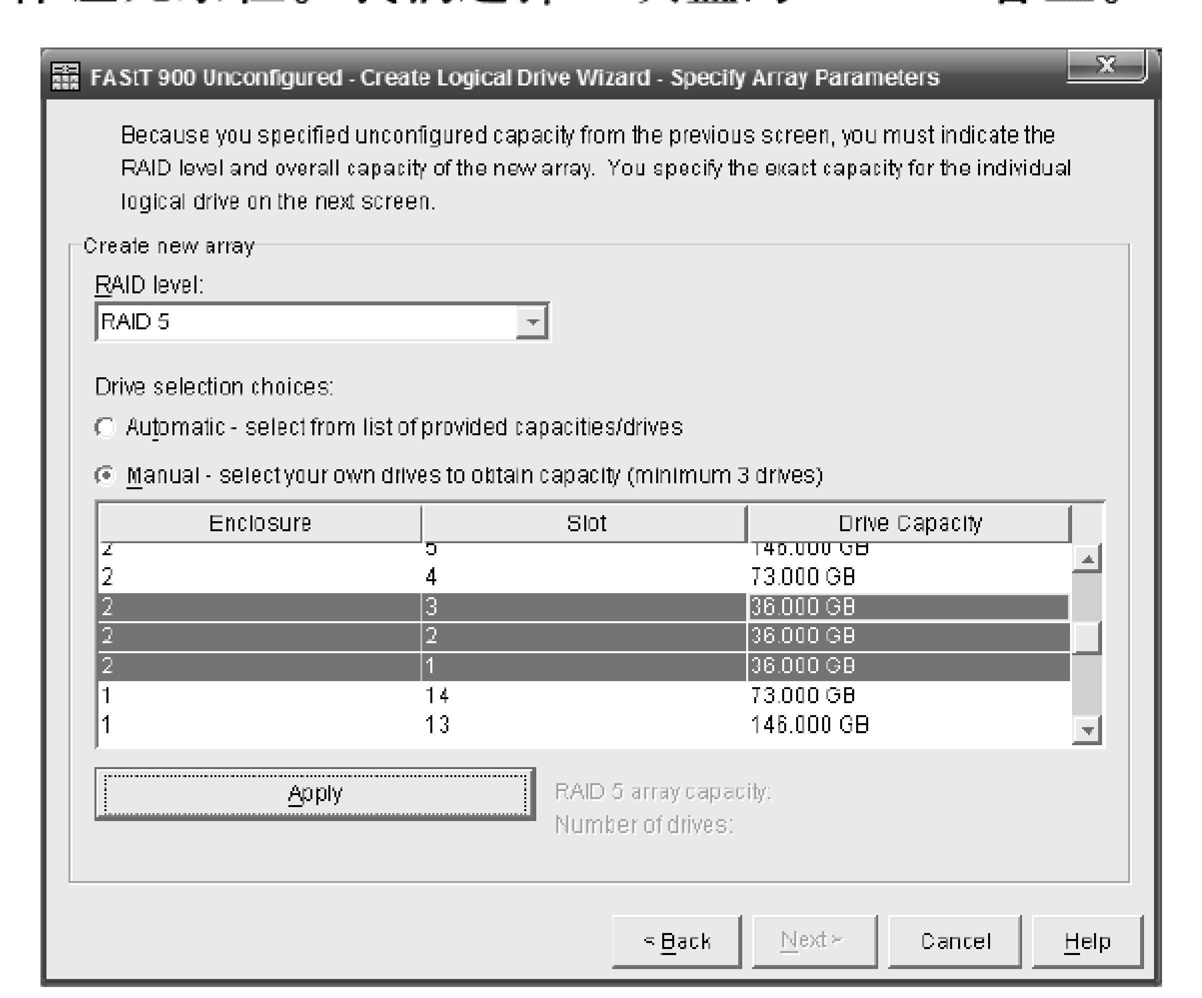


图9.60 RAID 类型选择，手动分配磁盘

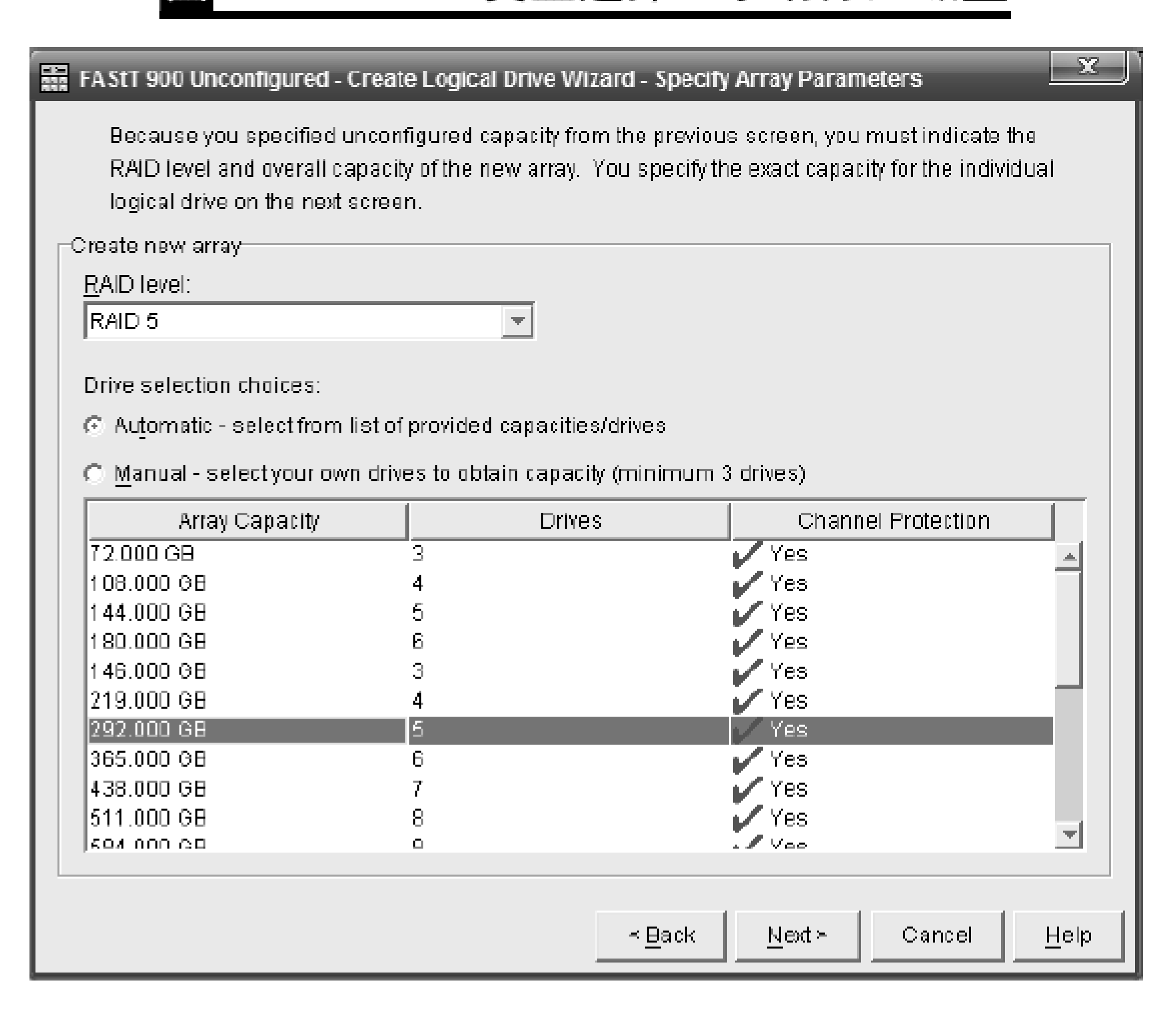


图9.61 自动分配磁盘

4】 在这个 RG 中，我们可以手动指定大小，划分出新的 LUN(即 Logical Drive)。这里将全部容量分配给一个 LUN，命名为 SQLLUN1，LUN 的参数选择自定义。

5】 选择条带深度为 32KB，每个条带的数据部分的宽度就是 32KB 乘以 4 块数据 Segment 为 128KB。这样，只要将 SQL Server 数据库的 Extent 参数设置为 128KB 大小，就可以保证每次读写 RG 均为整条读写，充分提高 IO 性能(但这样做会丧失并发 IO 能力)，如图 9.62 所示。Preferred controller ownership 表示这个 RAID Group 平时由哪个控制器管理。我们选择 Slot A，即控制器 A。一旦控制器 A 发生故障，这个 RG 会立即由控制器 B 接管。

6】 Logical Drive-to-LUN Mapping 即 LUN 映射，配置这个 LUN 将给哪个或者哪些主机使用。我们稍后选择用专门的配置模块配置，如图 9.63 所示。

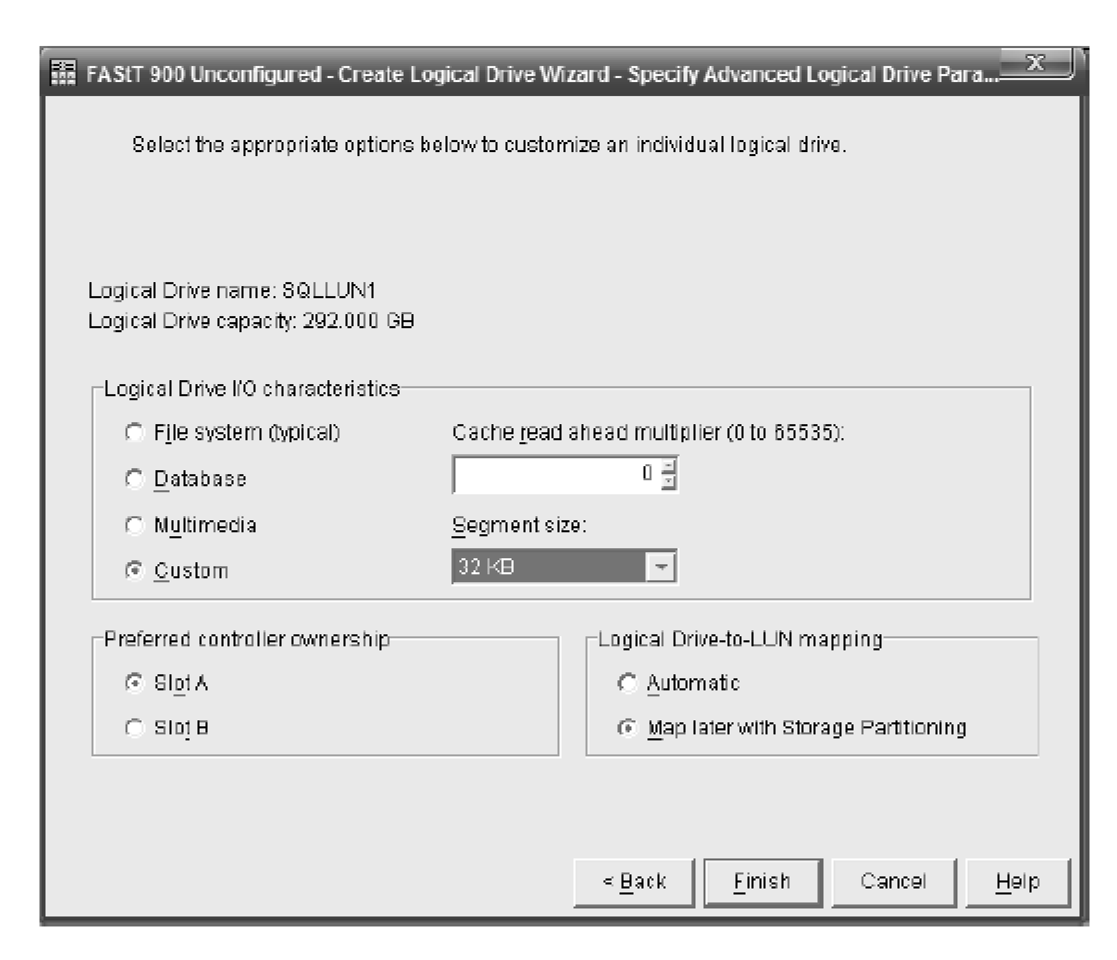


图9.62　选择条带深度

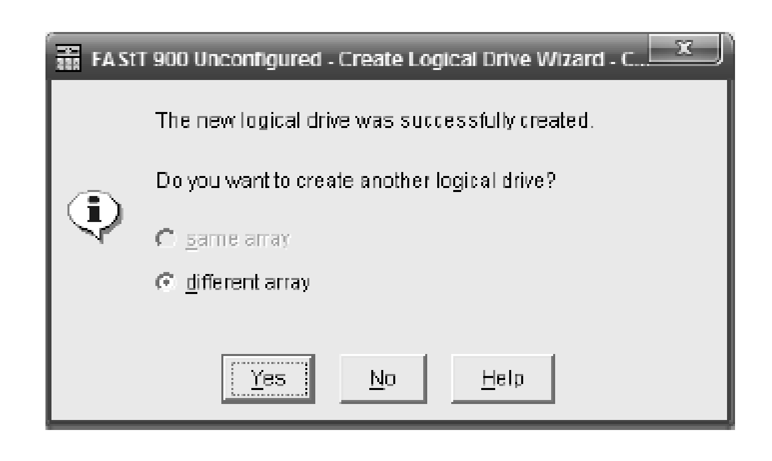


图9.63　配置 LUN 映射

7】 配置完后，系统会提示是否还要配置另外的 RG 或者 LUN。因为这个 RG 全部容量已经分配给了一个 LUN，所以窗口中的 same array 单选按钮为灰色的。单击 Yes 进入新一轮的 RG 和 LUN 配置。用这种方法，配置所有 6 个 LUN，如图 9.64 所示。

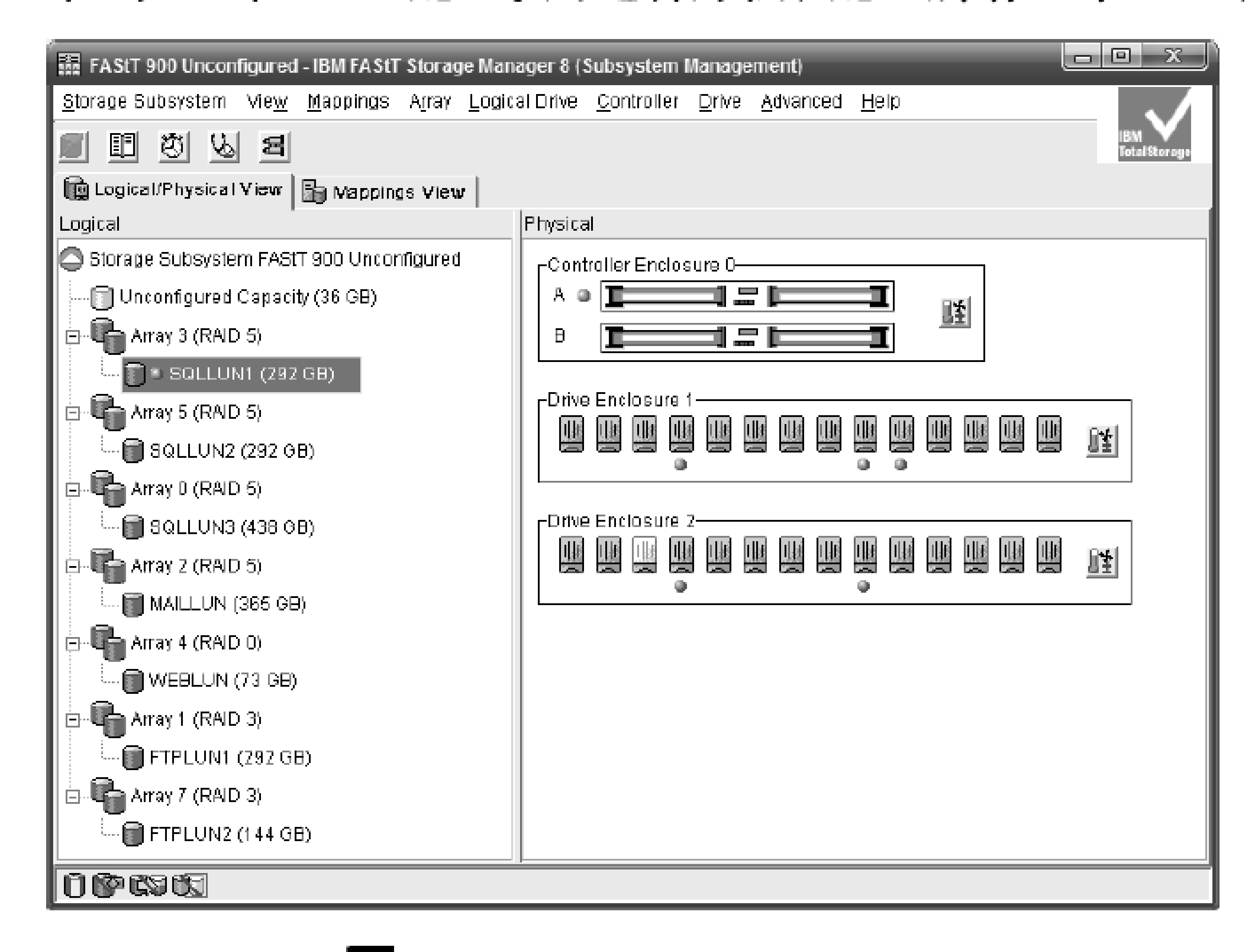


图9.64　创建完毕的 6 个 LUN

由于这个环境中的 FTP 服务器主要存放视频等大文件，所以使用 RAID 3 级别来提高传输性能，如图 9.64 所示。另外，只要选中某个创建好的 RG 或者 LUN，右边窗口中便会显示出这个 RG 和 LUN 所对应的物理磁盘以及掌管它的控制器。

2. 设置全局热备磁盘

1】 最好设置 1 个或者 2 个热备磁盘。这样，一旦某块磁盘损坏，热备磁盘立即顶替，以免数据丢失，如图 9.65 所示。

2】 设置完热备磁盘之后，可以进一步更改每个 RG 或者 LUN 的细节参数，如图 9.66 和图 9.67 所示。

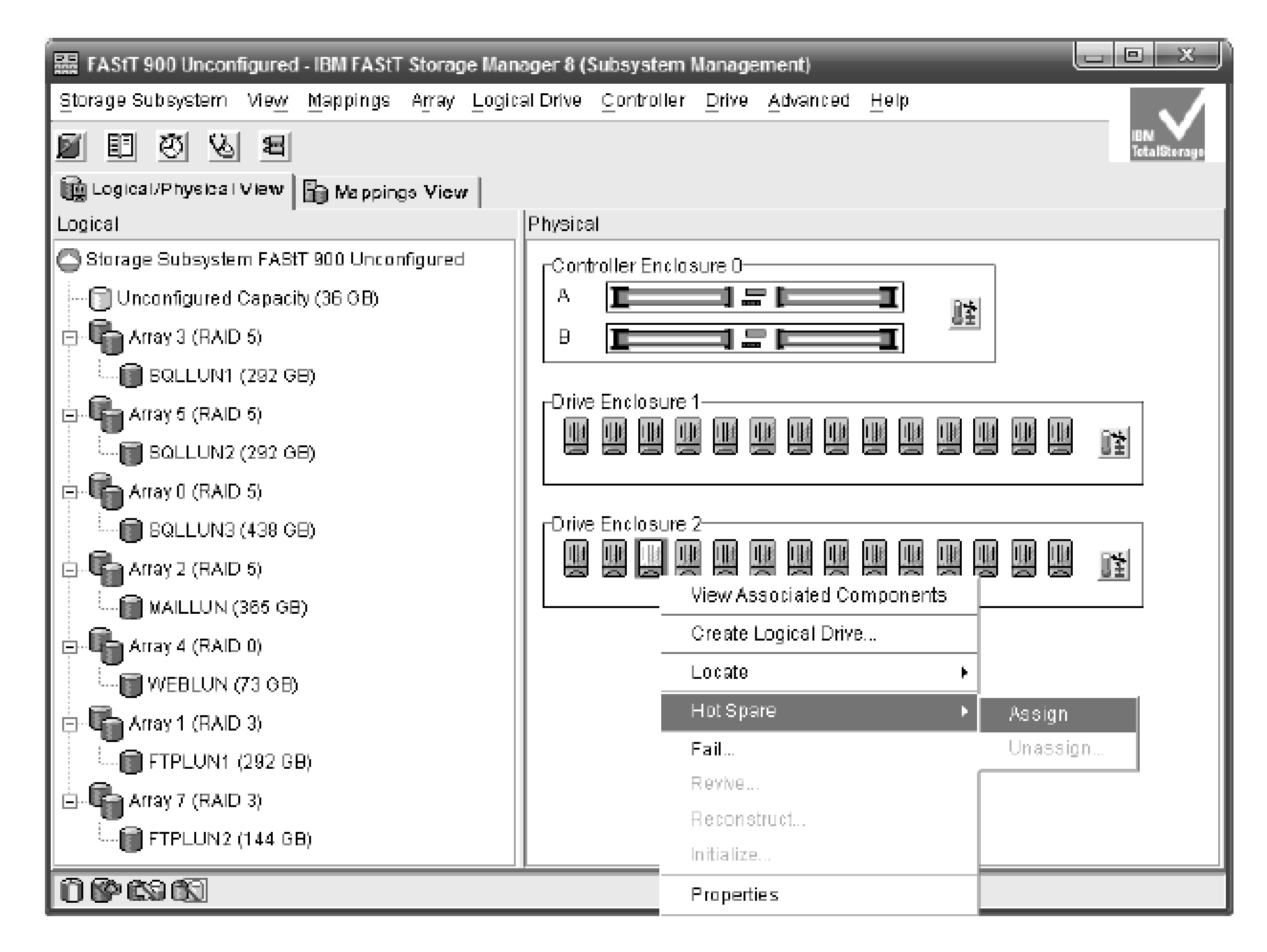


图9.65 选择热备盘

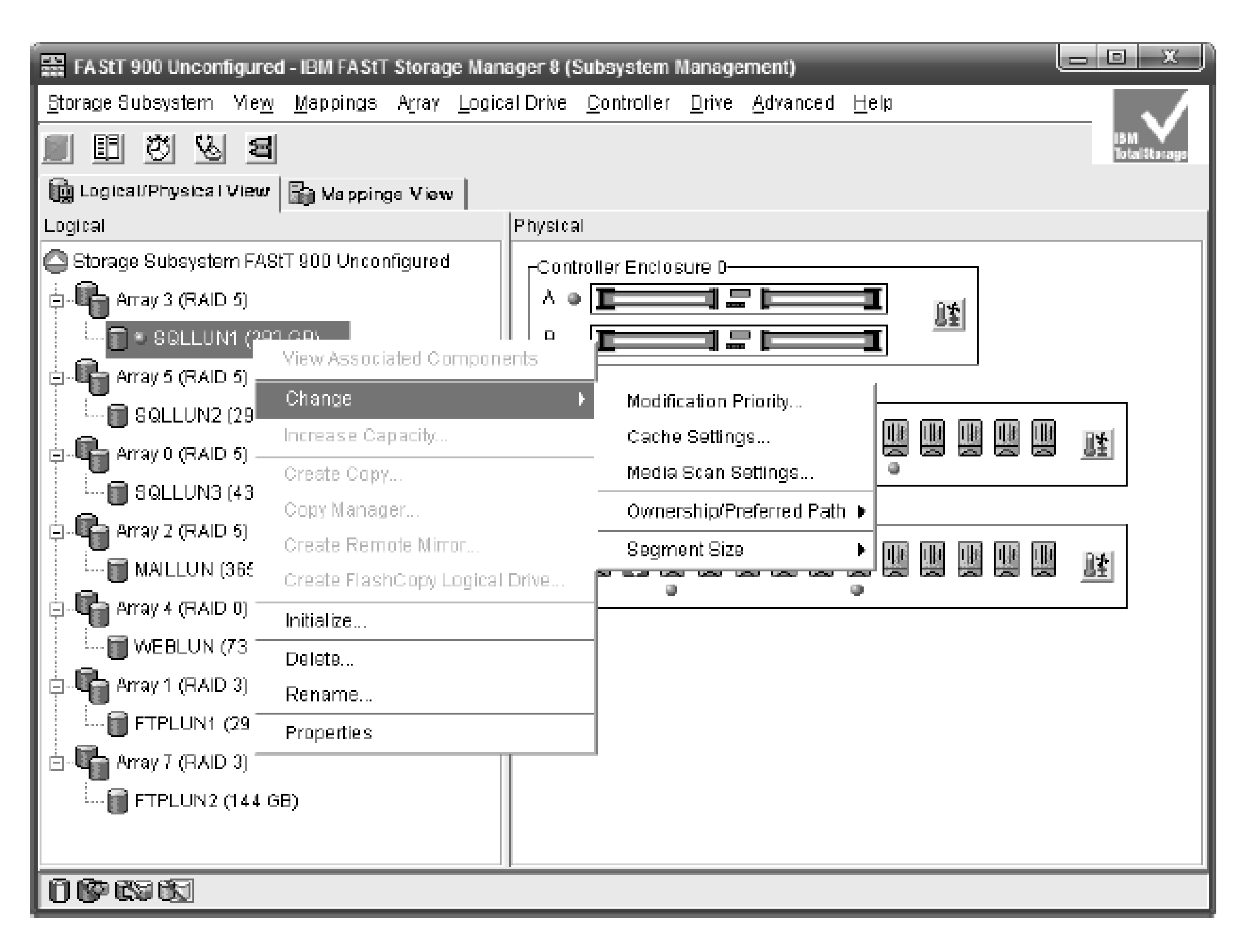


图9.66 各种参数(1)

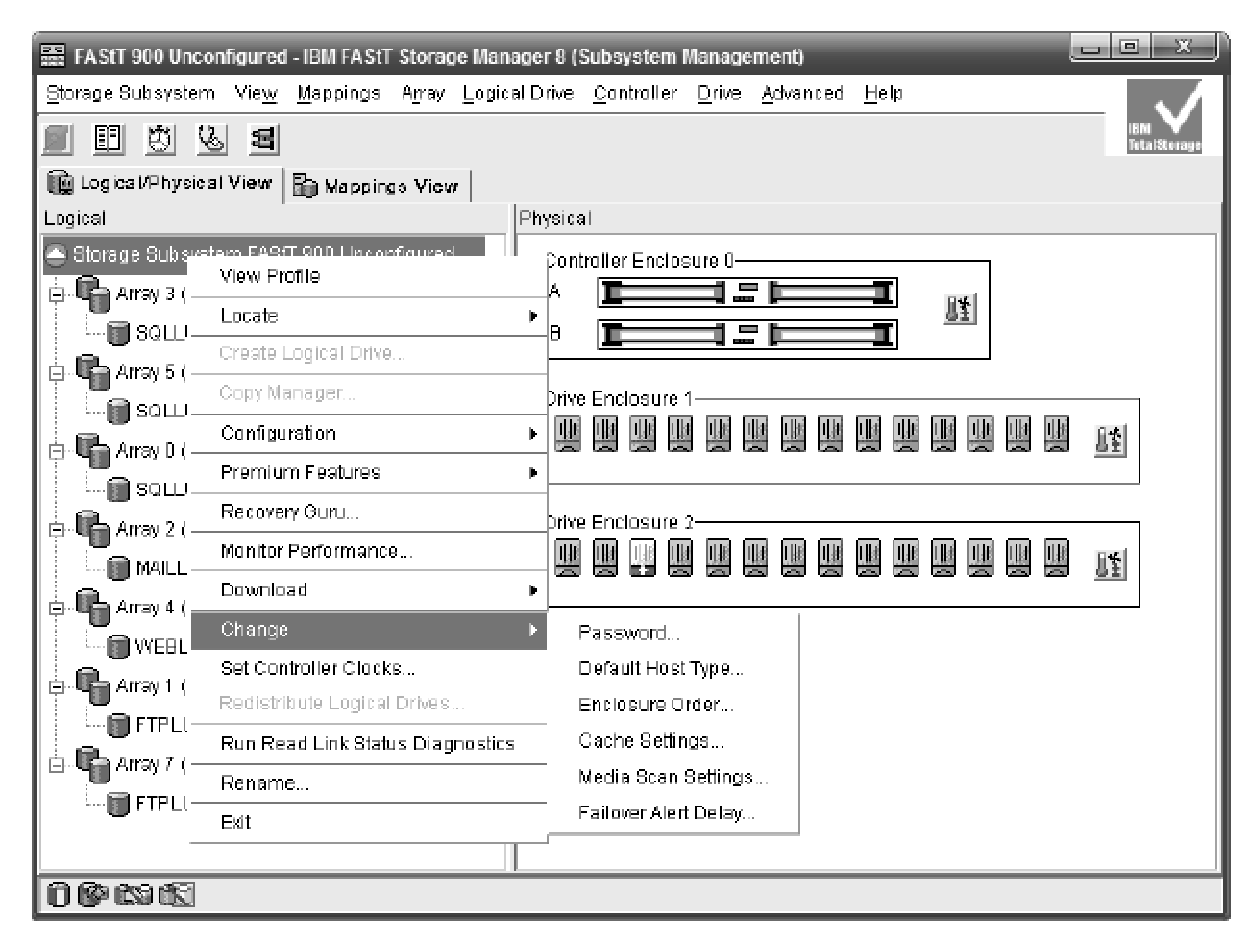


图9.67 各种参数(2)

3. 设置 LUN 映射

1】 切换到主界面的 Mappings View 选项卡，然后右击设备名，选择 Define Host 命令来添加主机信息，即使用这台盘阵的主机信息，如图 9.68 所示。

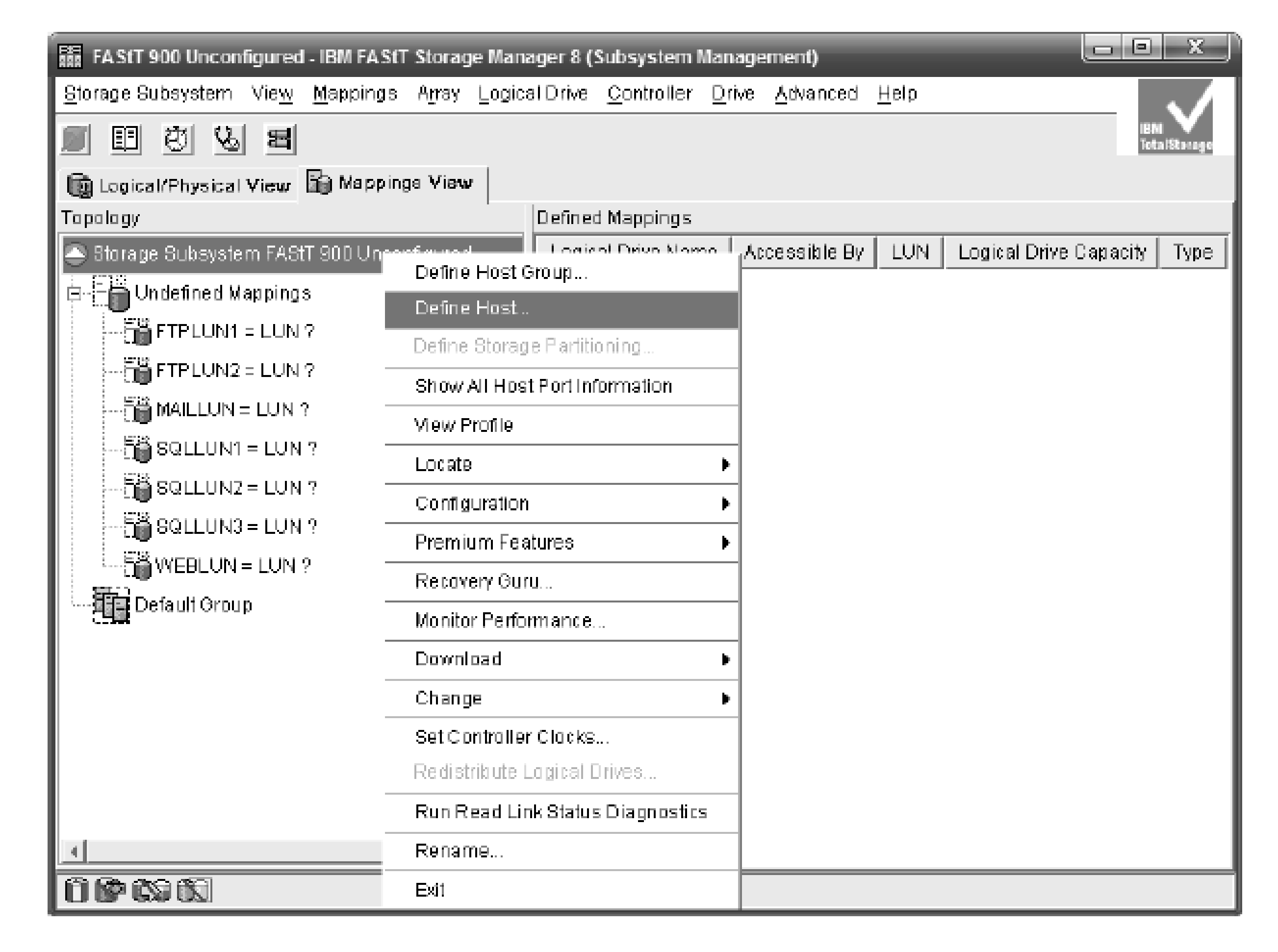


图9.68 添加主机映射

2】 Host Name 可以随便起名，并不一定要与对应主机的真正 Hostname 相同。添加完所有主机后，界面的显示如图 9.69 所示。

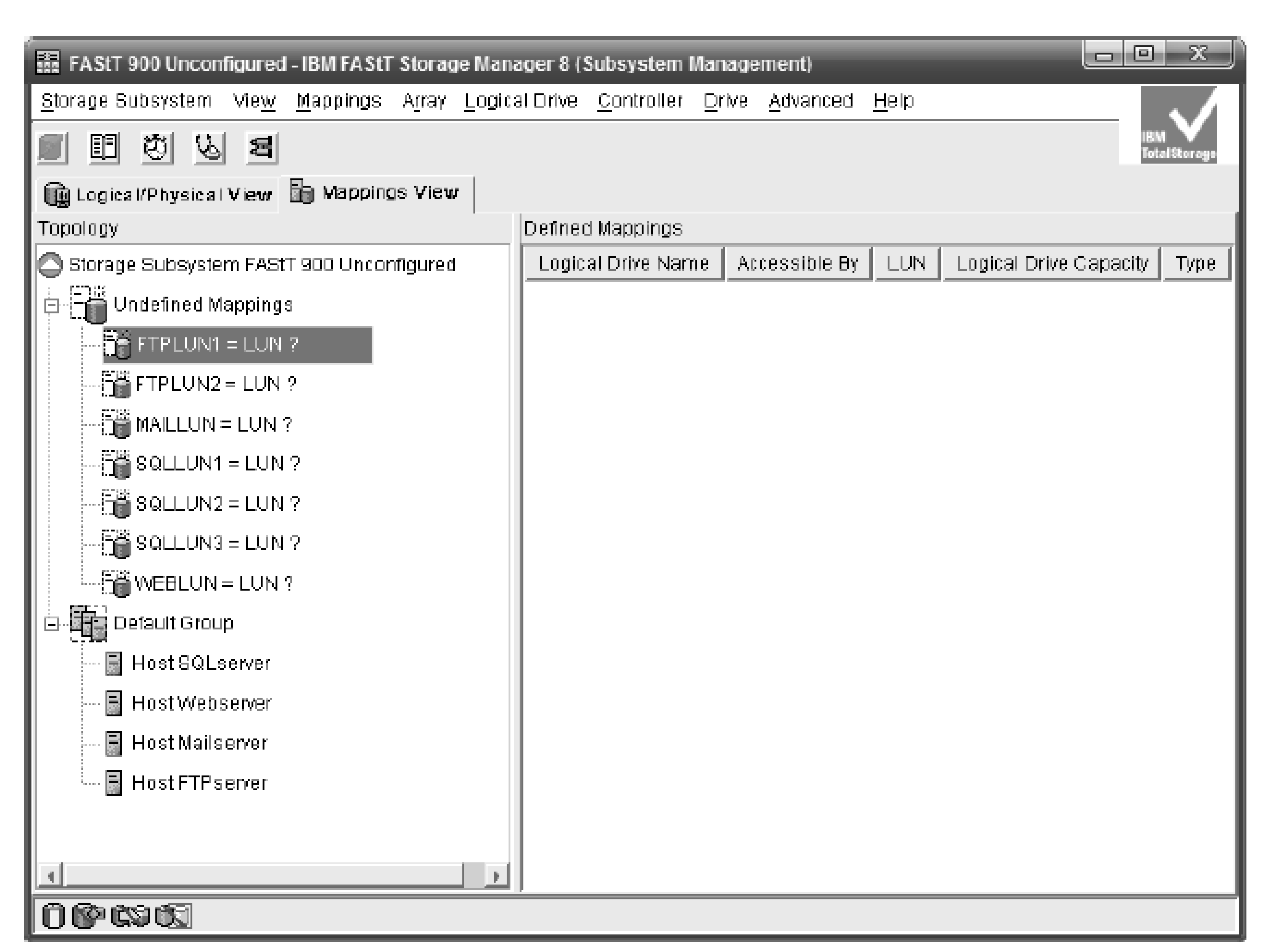


图9.69 主机列表

3】 添加完主机名之后，还必须添加 Host Port，即必须让盘阵知道主机使用哪些 FC 接口来访问盘阵。右击某个主机名，选择 Define Host Port 命令，如图 9.70 所示。

4】 在 Port Identifier 处，我们选择对应的 ID(即每个主机 FC HBA 卡上的 WWN 地址，盘阵会通过 Fabric 网络自动发现这些地址)。主机类型当然选择 Windows 2000/Server 2003 Non-C，Host port name 随便起一个名字即可，如图 9.71 所示。每台服务器上有两块 FC 卡，所以每台主机共有两个接口，在此为每个接口都定义一下。定义好每台主机的接口之后，如图 9.72 所示。

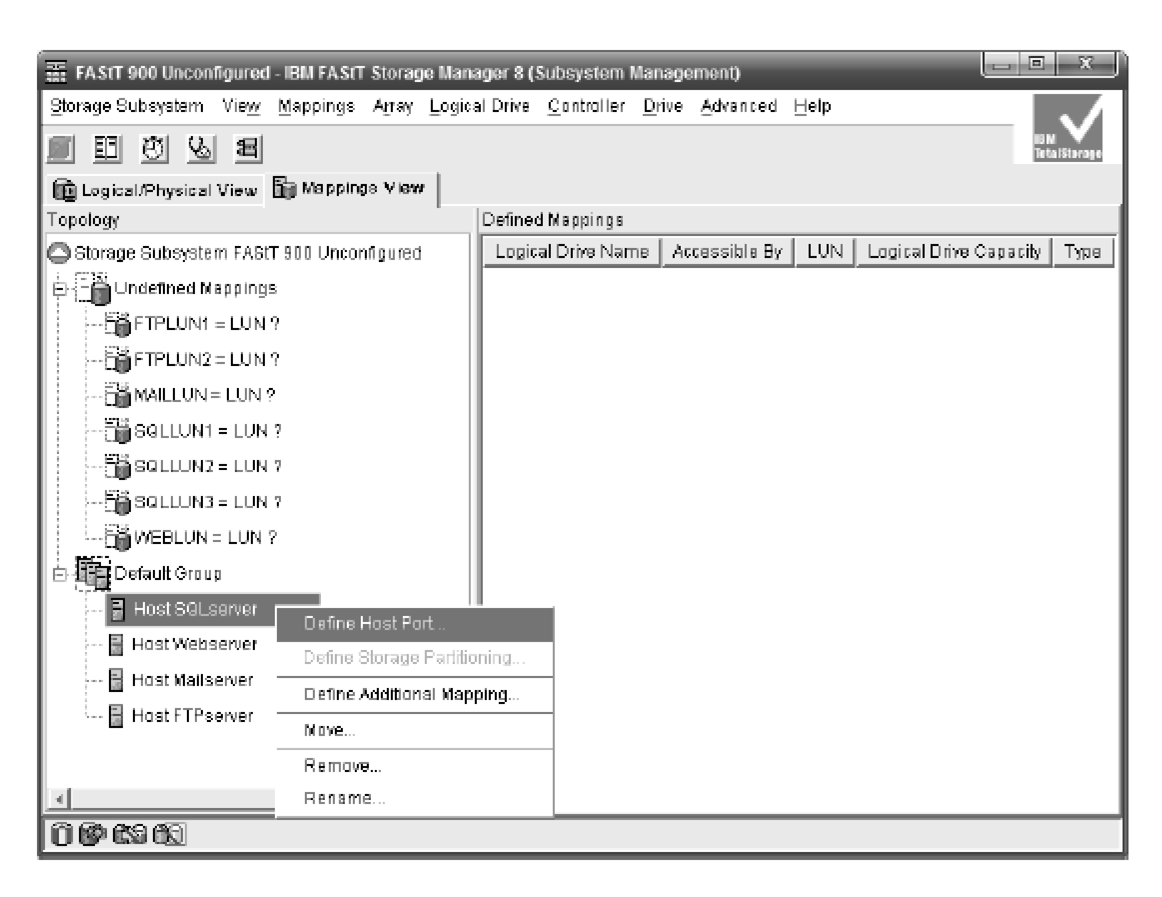


图9.70 添加主机的 FC 端口信息

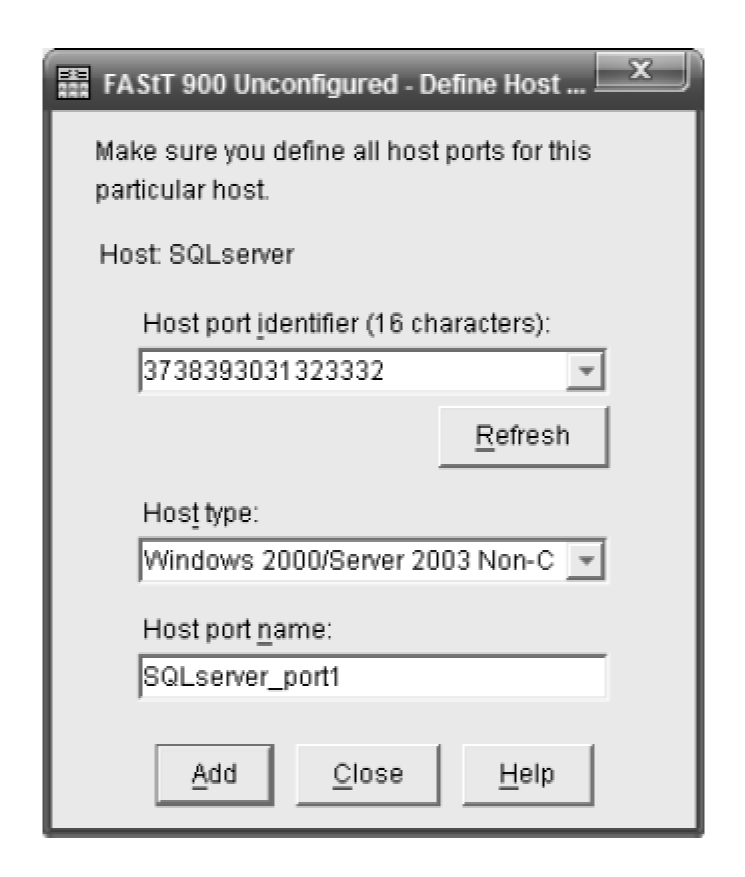


图9.71 主机上的 FC 端口信息

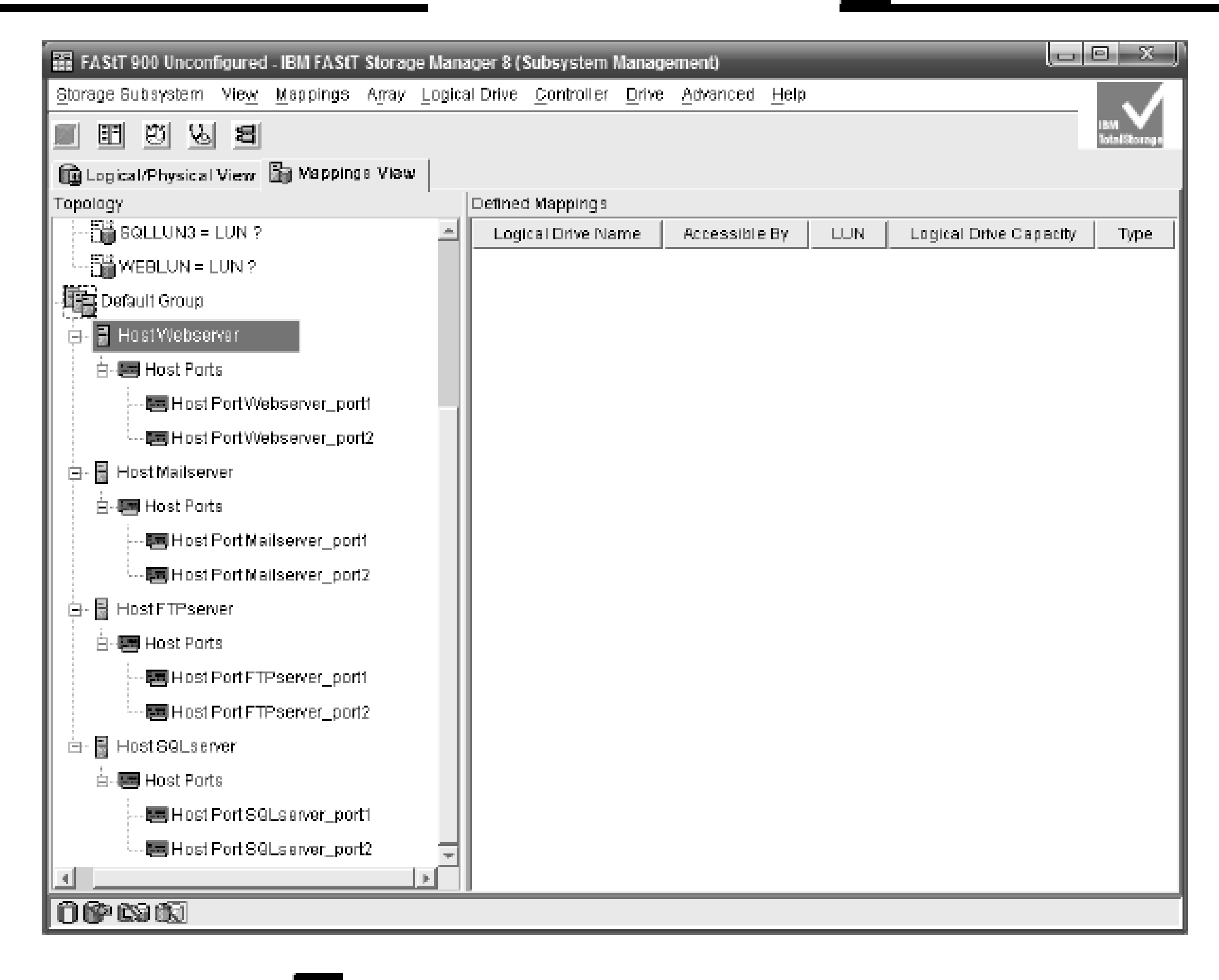


图9.72 所有主机端口定义完毕

5】 在未映射的 LUN 上右击，选择 Define Additional Mapping 命令，打开如图 9.73 所示的对话框。

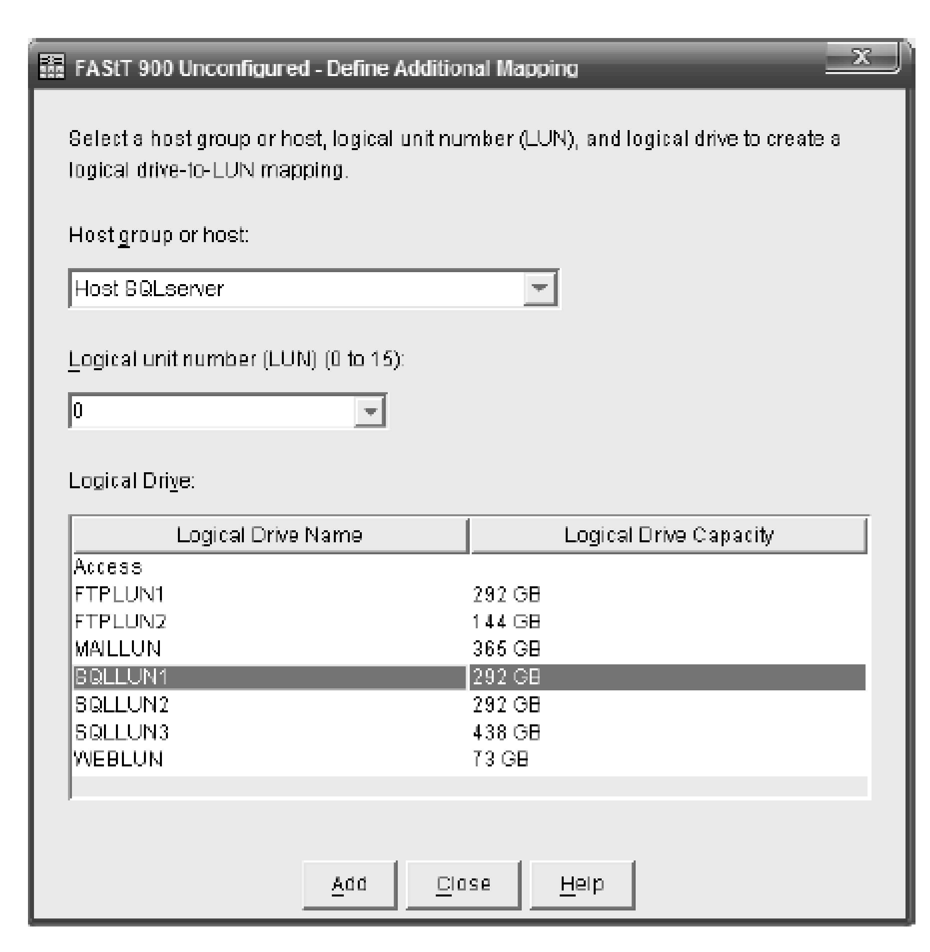


图9.73 映射 LUN 到主机端口

6】 在 Host group or host 下拉列表框中选择 Host SQLserver 这台主机，Logic unit number 表示此次映射给这台主机的 LUN，在主机上将显示 LUN 的号码。图 9.73 中，将 SQLLUN1 这个 LUN 映射给主机 SQLserver，主机上显示的对应这块 LUN 的号码是 LUN0。用这种方法，将所有属于 SQLserver 主机的 3 个 LUN 都映射好后，单击 close。结果如图 9.74 所示。

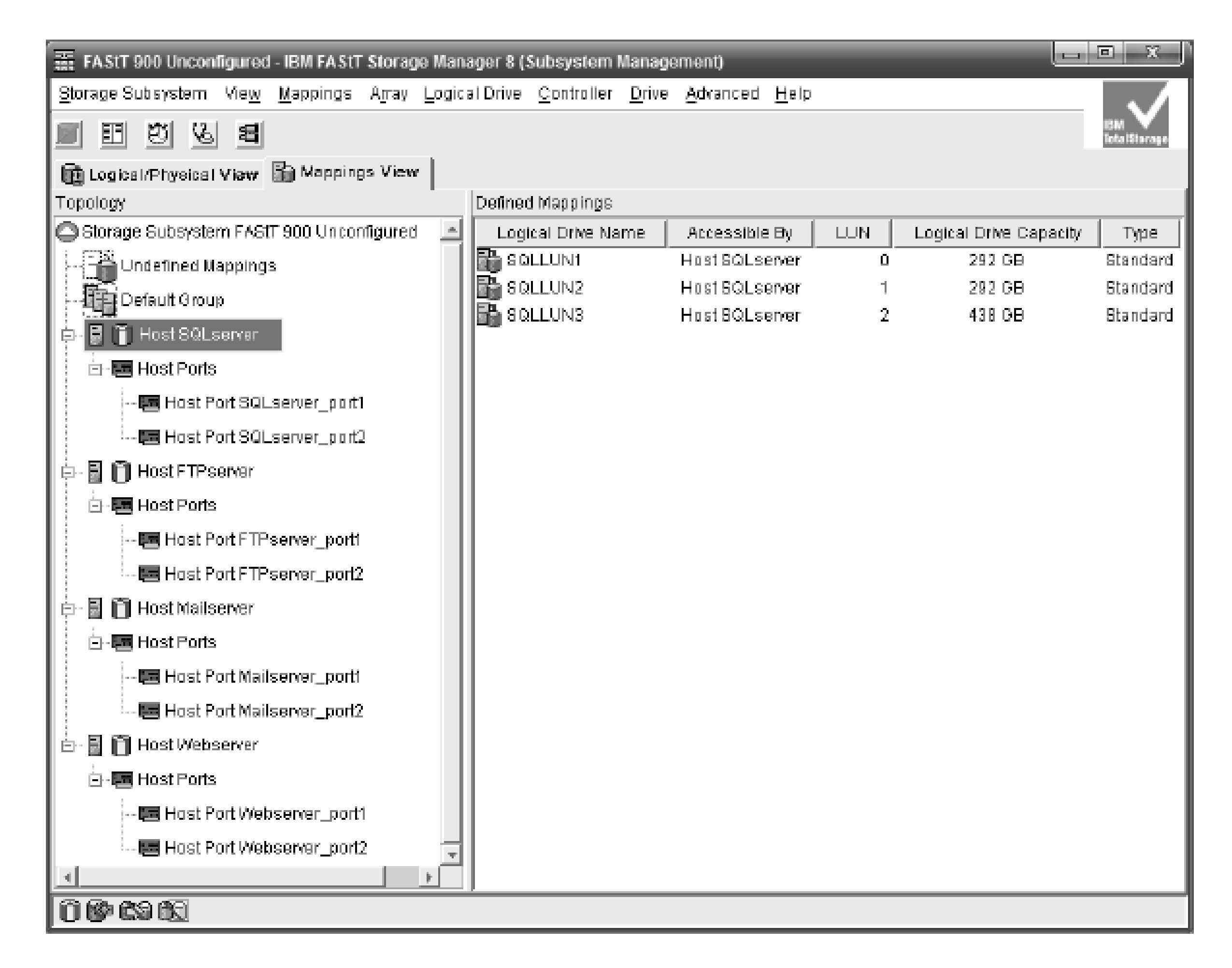


图9.74　LUN 映射完毕



单击每台主机，右边就会出现这台主机可以访问的 LUN 及其相关信息。注意：每个 LUN 只能映射给一台主机。理论上，一个 LUN 完全可以映射给多台主机共同访问使用的，这就需要用到 partition 功能了。这里就不做过多介绍了。

4. 初始化 LUN 配置

1】 所有设置做完之后，需要进行初始化。初始化完毕之后，才能供主机使用。

2】 右击每个 Array 或者 LUN，然后选择 Initialize 命令，如图 9.75 和图 9.76 所示。

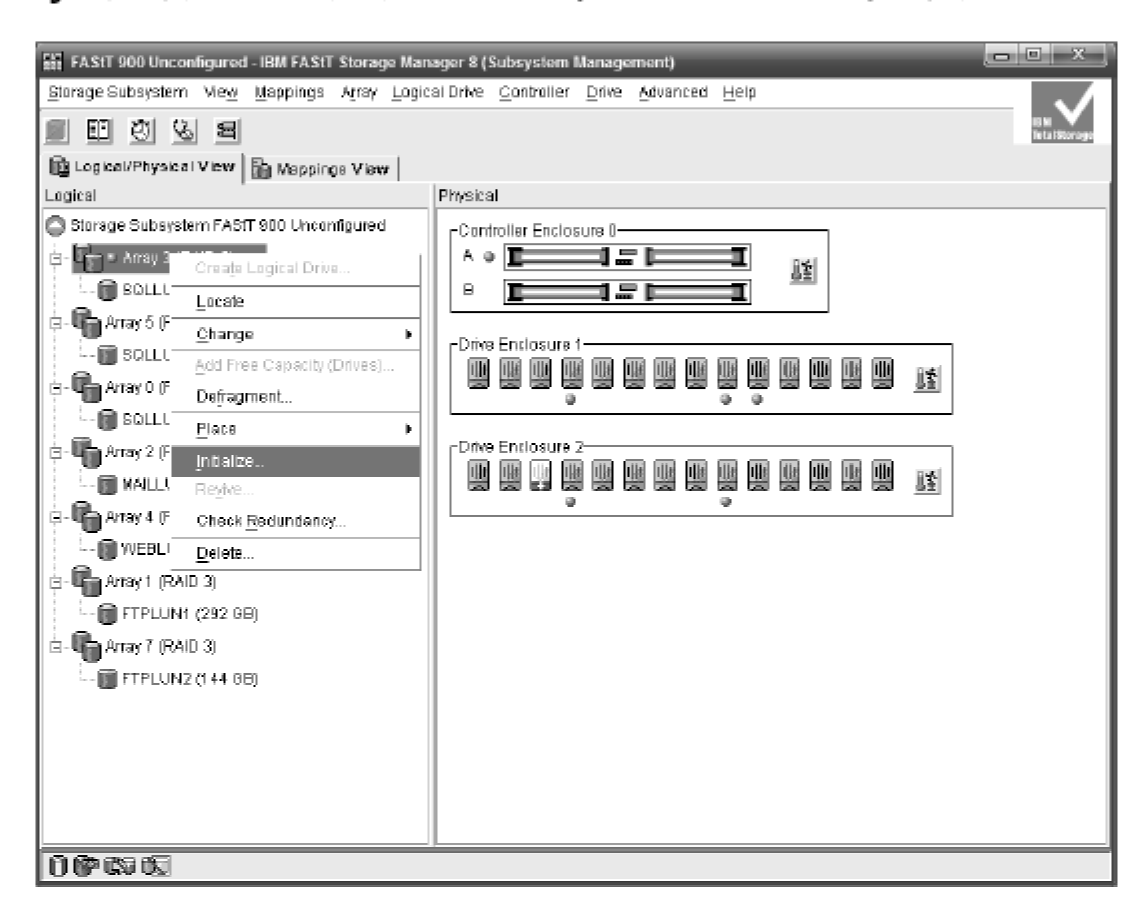


图9.75　初始化磁盘阵列

3】 右击每个 Array，选择属性命令，可以查看初始化的进度，如图 9.77 所示。

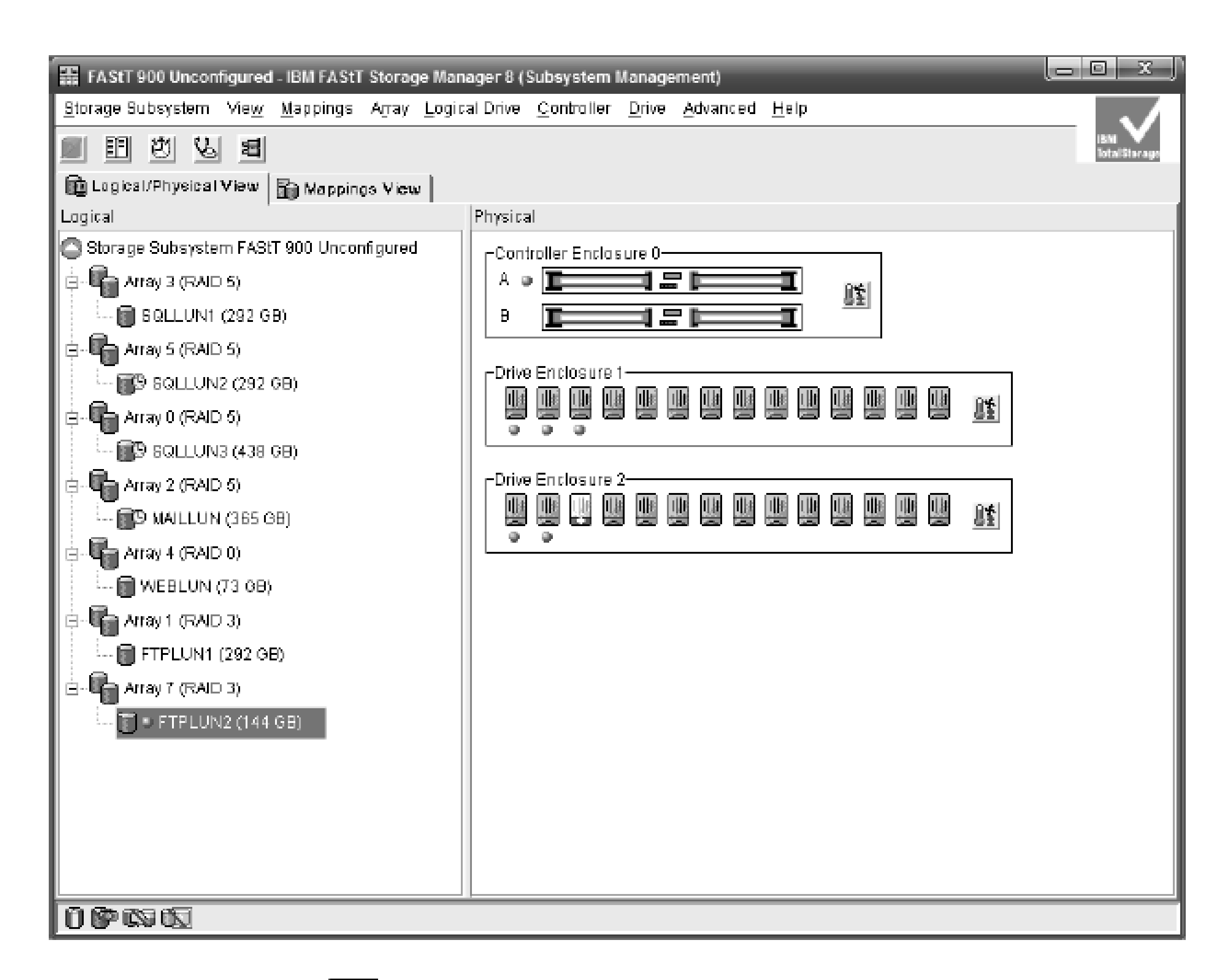


图9.76　初始化所有 RAID 组

FAStT 900 Unconfigured - Logical Drive Properties

Logical Drive name: SQLLUN3
Logical Drive ID: 32:33:34:35:36:37:38:39:30:31:32:36:35:00:00:00:00:00:00:00
Subsystem ID (SSID): 10
Status: Optimal
Preferred owner: Controller in slot A
Current owner: Controller in slot A
Capacity: 438 GB
RAID level: 5
Segment size: 32 KB
Modification priority: Low
Associated array: 0
Read cache: Enabled
Write cache: Enabled
Write cache without batteries: Disabled
Write cache with mirroring: Enabled
Flush write cache after (in seconds): 5.00
Cache read ahead multiplier: 5
Enable background media scan: Enabled
Media scan with redundancy check: Disabled
Initialization - Progress
NOTE: To change the rate of modification, use the Logical Drive>>Change>>Modification Priority option.
Close　Help

图9.77　初始化进度窗口

初始化完成之后，盘阵即可接受主机访问了。

其他不同品牌型号的磁盘阵列产品的配置过程大同小异，只要理解了磁盘阵列的组成架构和原理，配置起来其实是很简单的。

9.5.2　基于 EMC 的 CX700 磁盘阵列配置实例

在此我们简要介绍一下 EMC 针对 CX 系列的配置工具 Navisphere。

1 登录 Navisphere，查看全局情况

全局视图如图 9.78 所示。图 9.79 为所连接的两个扩展柜中的一个柜子里的磁盘列表。图 9.80 所示为控制器柜里包含的两个控制器的信息。图 9.81 所示为存放盘阵操作系统及重要配置信息的私有 LUN。每个控制器都有各自的私有 LUN。图 9.82 所示为映射 LUN 与主机用的 Storage Groups，也就是映射组。在这个组中的主机可以访问这个组中的 LUN。

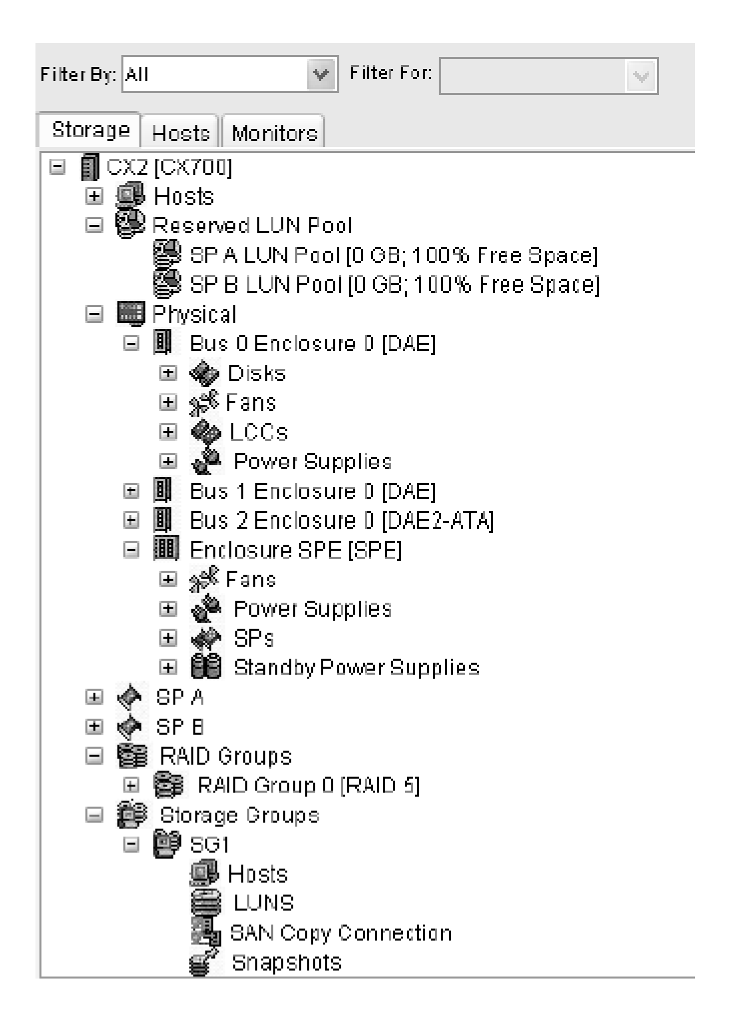


图9.78　全局视图

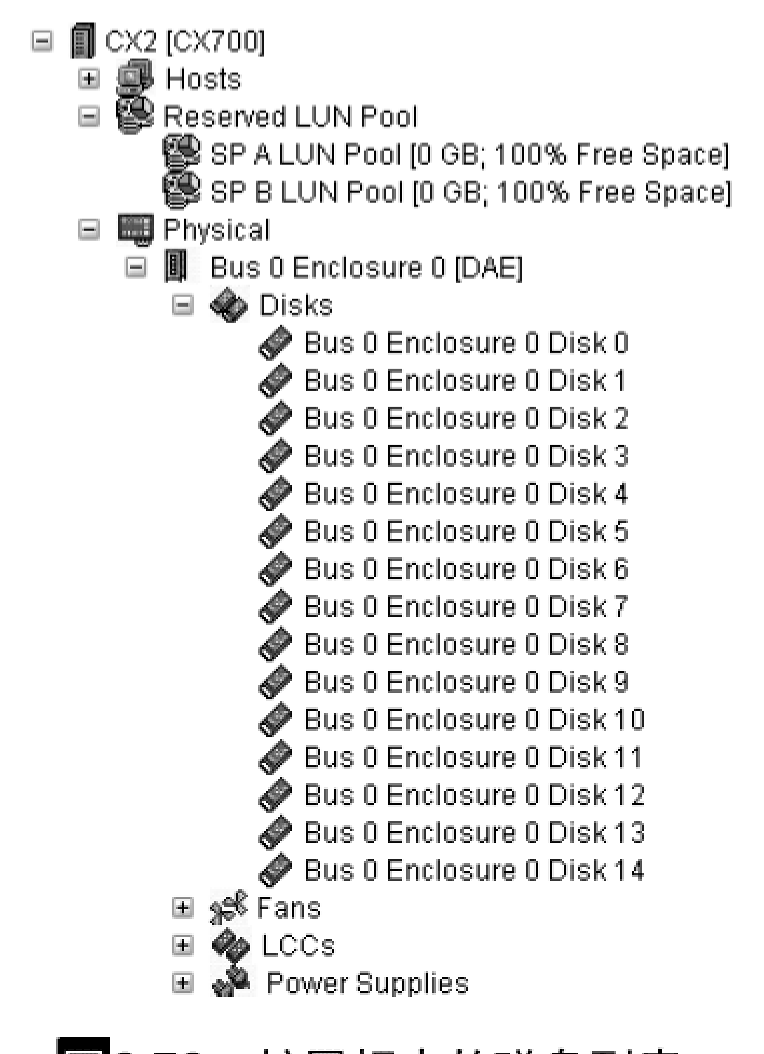


图9.79　扩展柜中的磁盘列表

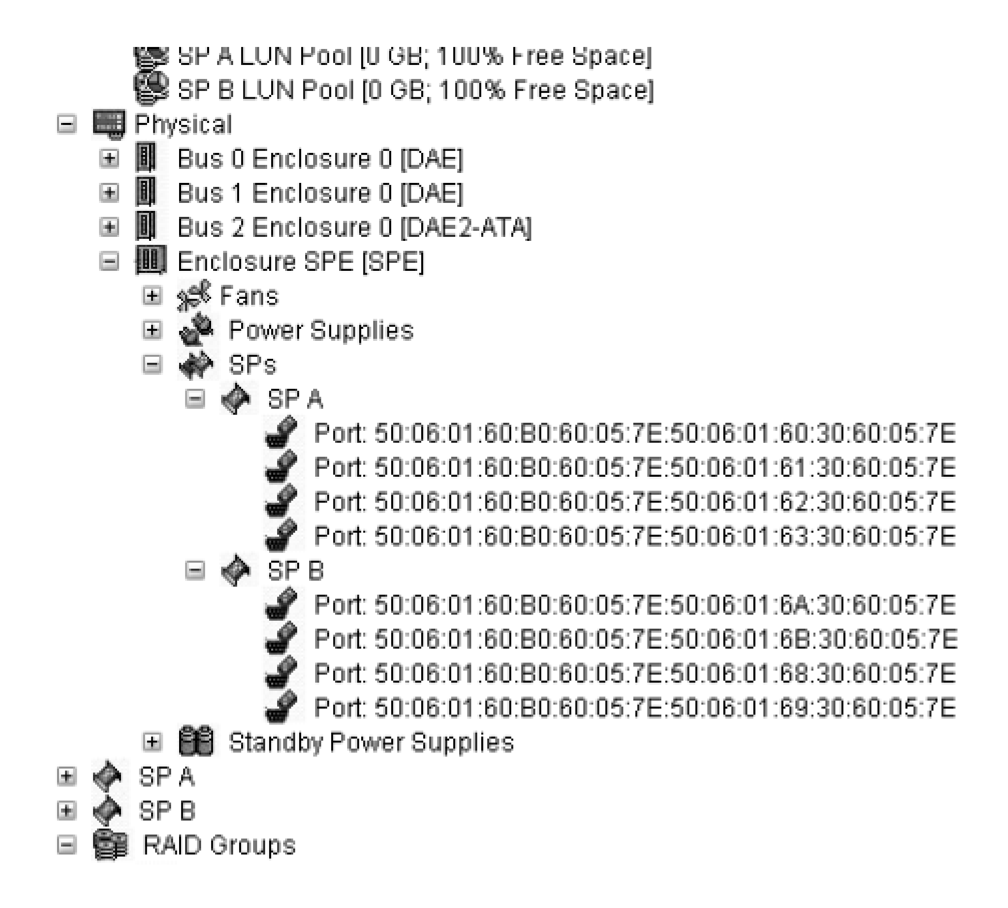


图 9.80 控制器柜中的两个控制器

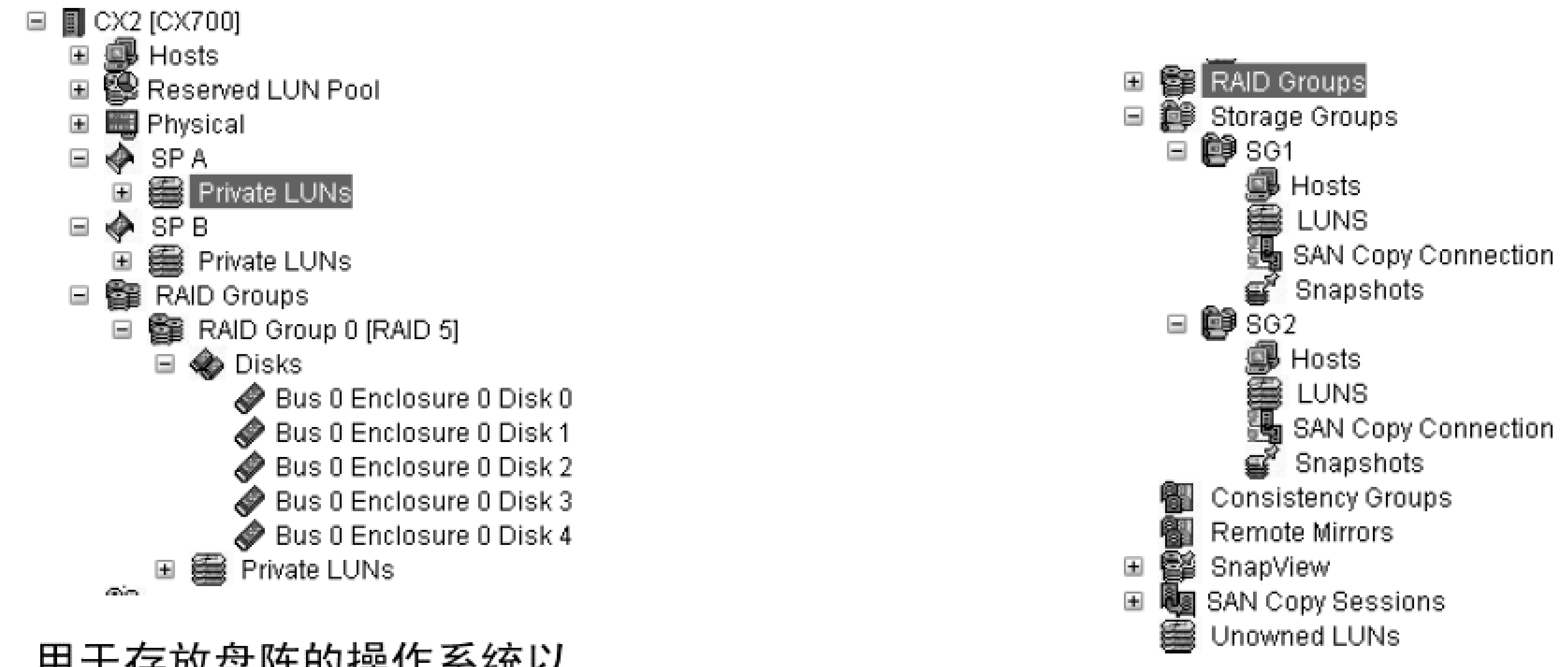


图9.81 用于存放盘阵的操作系统以及其他配置信息的 Private LUNs

图9.82 用来映射 LUN 和主机的 Storage Groups

2. 创建 RAID 组

在盘阵图标上右击，选择 Create RAID Group 命令，如图 9.83 所示。

在如图 9.84 所示的对话框中选择 RAID 组的 ID、RAID 组所包含的磁盘数量以及其他参数。

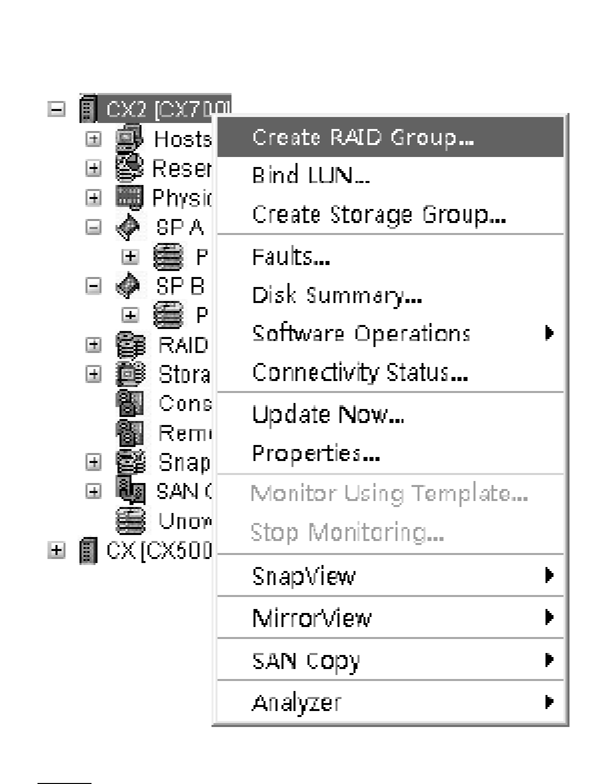


图9.83 创建 RAID 组

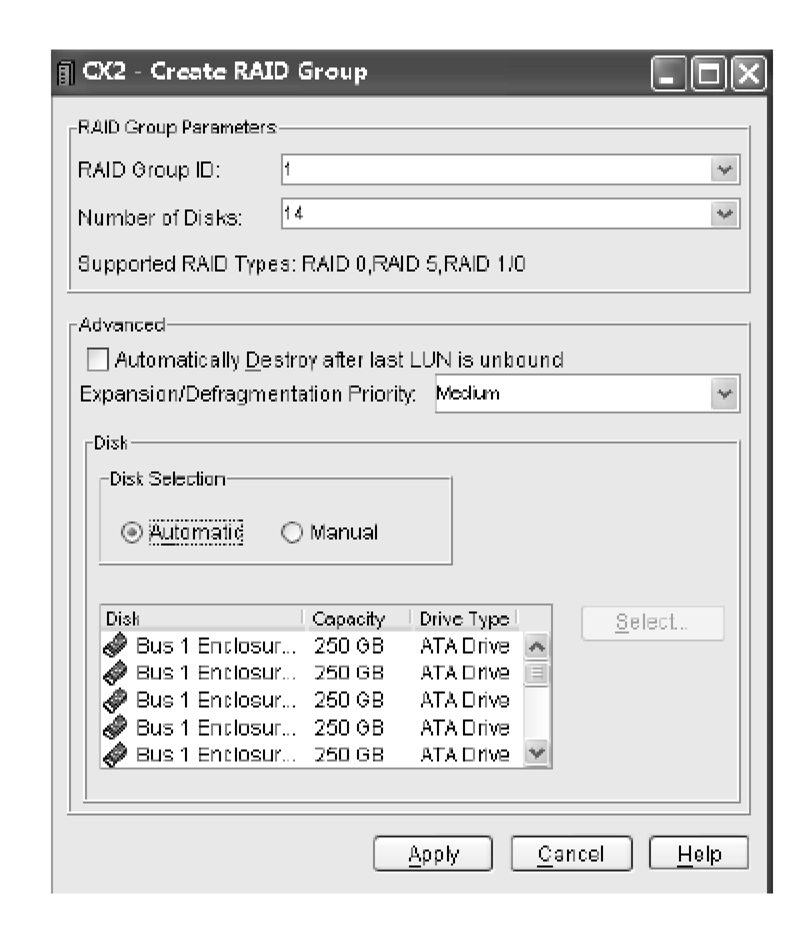


图9.84 RAID 组参数

在创建好的新 RAID 组上右击，选择 Bind LUN 命令创建 LUN，如图 9.85 所示。

在打开的对话框中，我们选择 RAID 类型为 RAID 5，如图 9.86 和图 9.87 所示。
利用上述方法创建 LUN 6 和 LUN 7 两个 LUN，如图 9.88 所示。

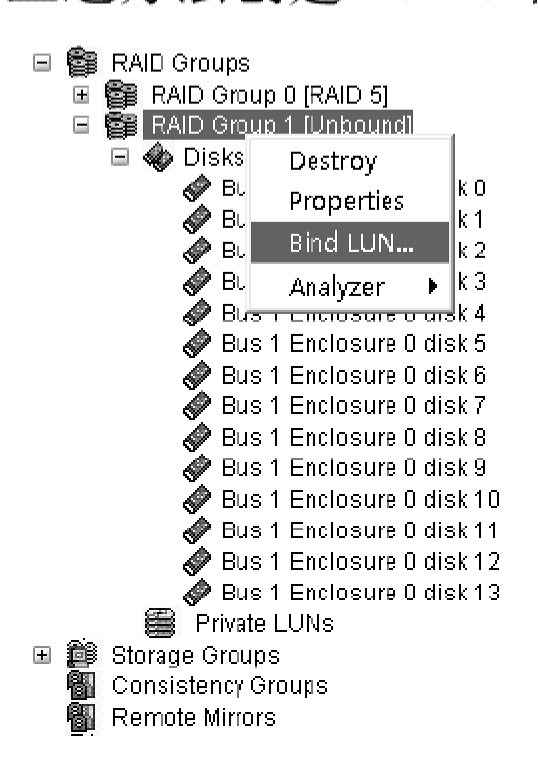


图9.85 在 RAID 组上创建(绑定)LUN

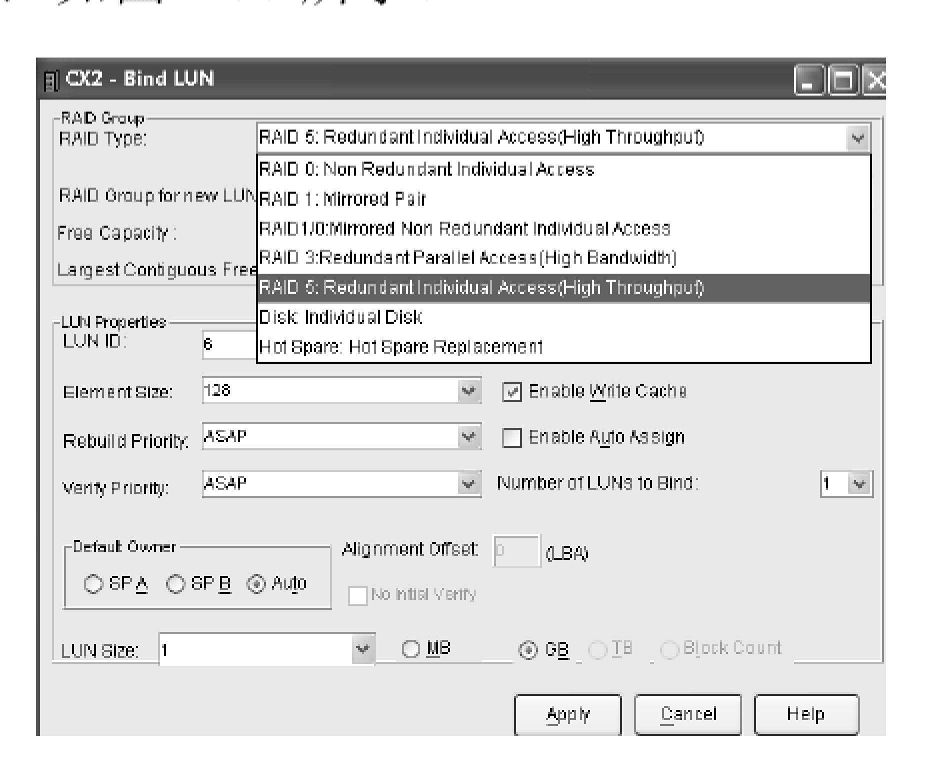


图9.86 LUN 参数页(1)

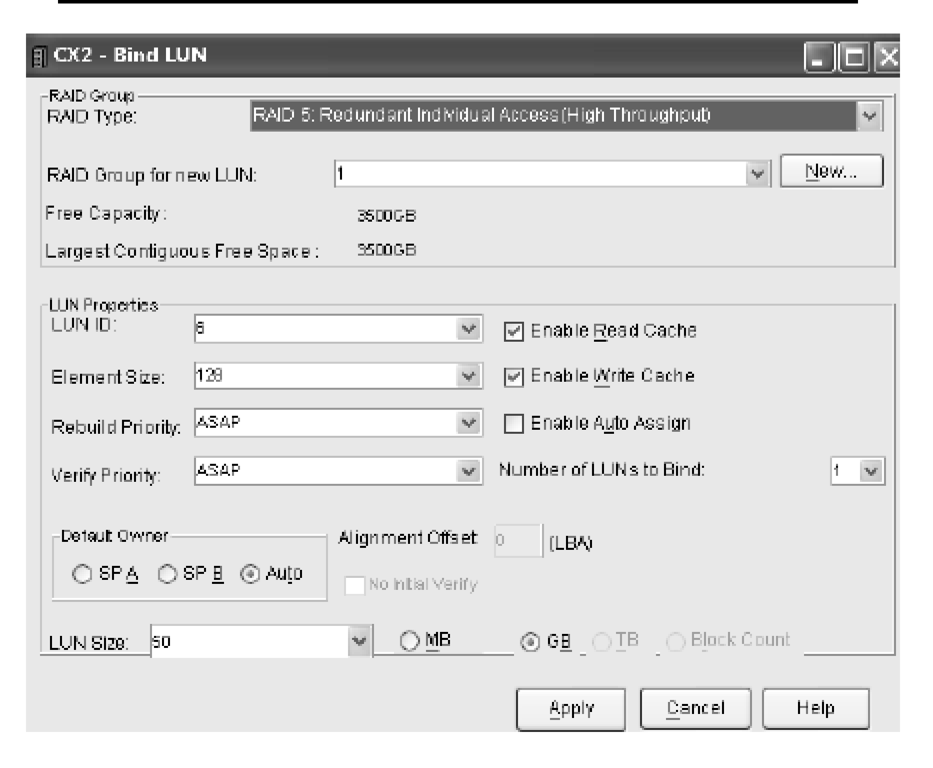


图9.87 LUN 参数页(2)

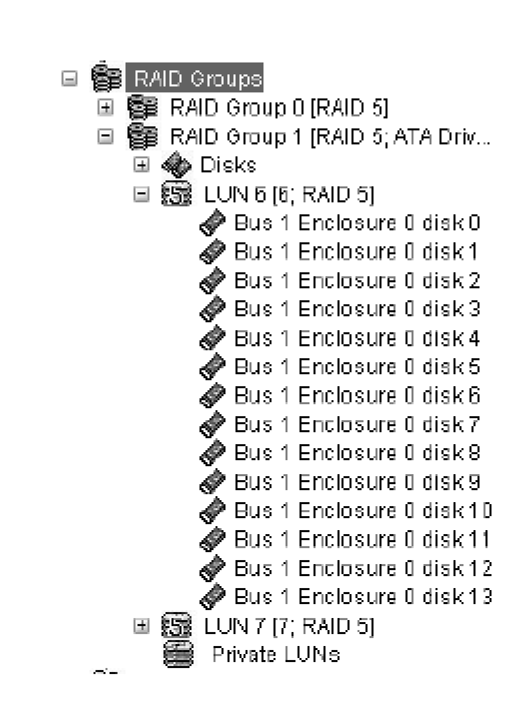


图9.88 创建好的两个 LUN：LUN 6 和 LUN 7

3. 创建 Storage Group 并绑定主机与 LUN

在盘阵图标上右击，选择 Create Storage Group 命令，如图 9.89 所示。
在打开的对话框中输入新 Storage Group 的名称，如图 9.90 所示。

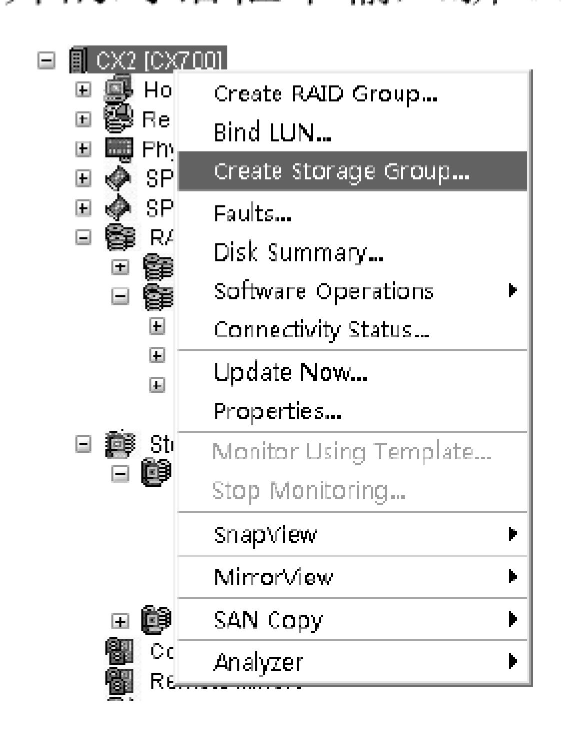


图9.89 创建 Storage Group

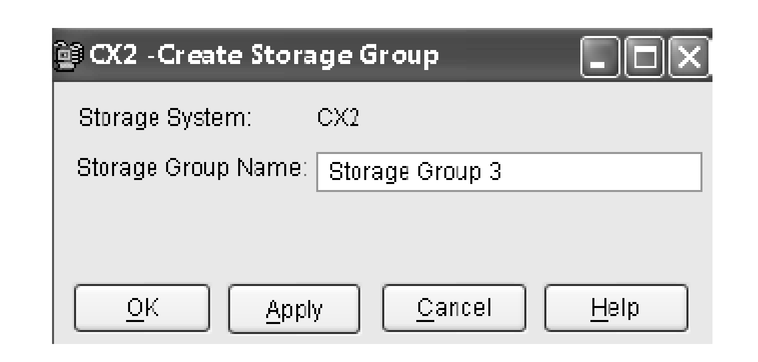


图9.90 给新 Storage Group 起名

在创建好的 Storage Group 上右击，选择 Select LUN 命令绑定 LUN，如图 9.91 所示。
将要分配给这个组的 LUN 移动到右边的窗口，如图 9.92 所示。
切换到 Hosts 选项卡，选择要分配的主机，将其移动到右侧窗口，如图 9.93 所示。

添加完 LUN 和主机之后，对应的主机就可以识别到对应的 LUN，并可以使用了，如图 9.94 所示。

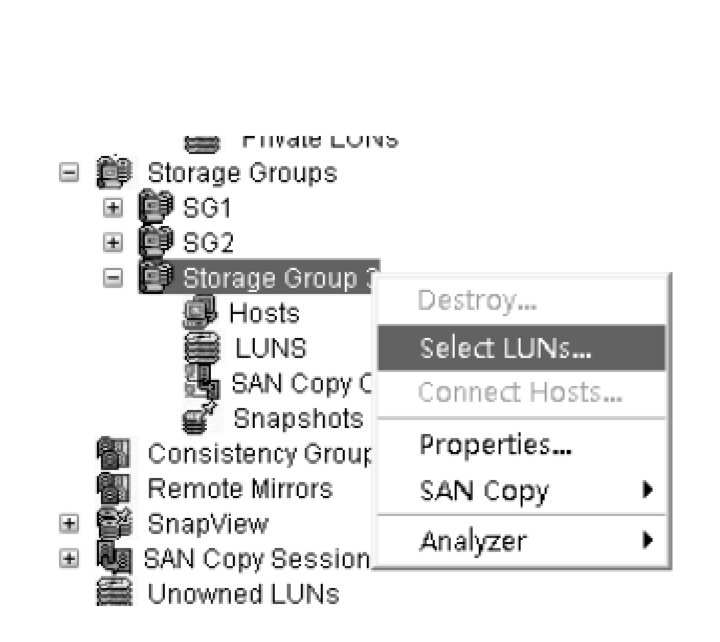


图9.91　选择要映射的 LUN

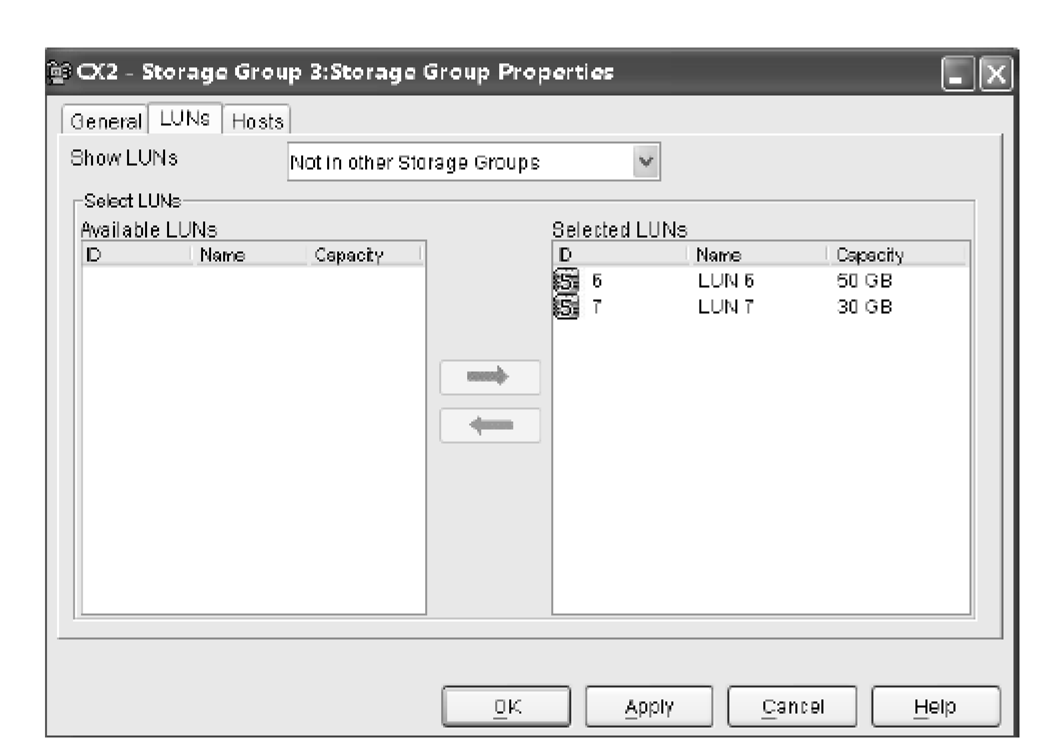


图9.92　选择要映射的 LUN

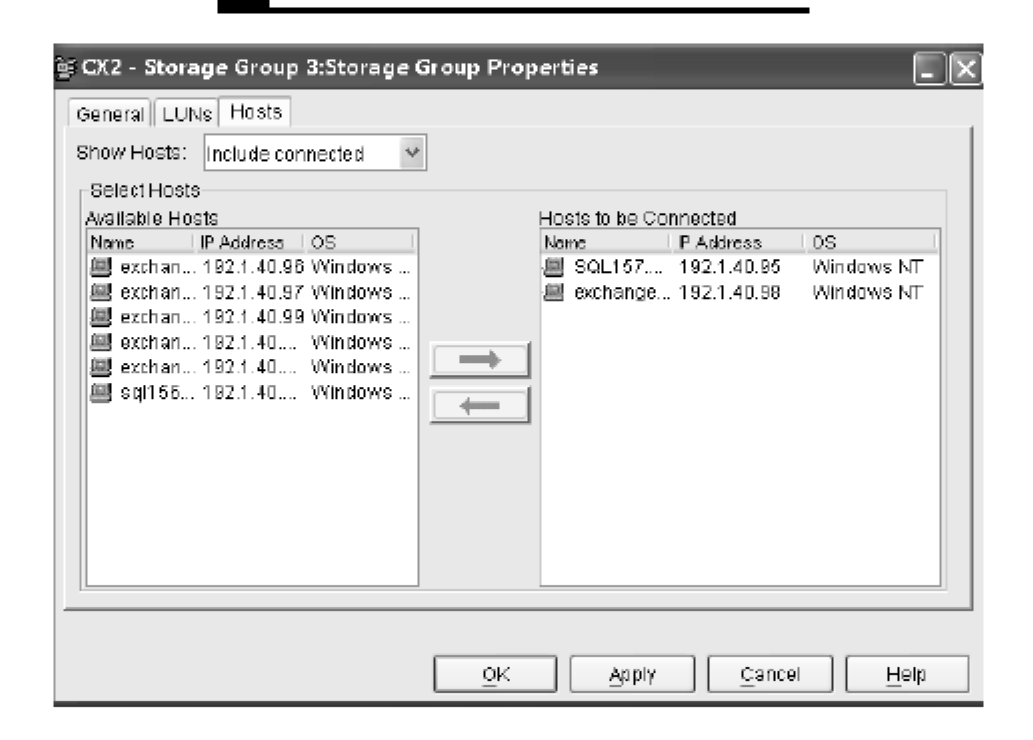


图9.93　选择要绑定的主机

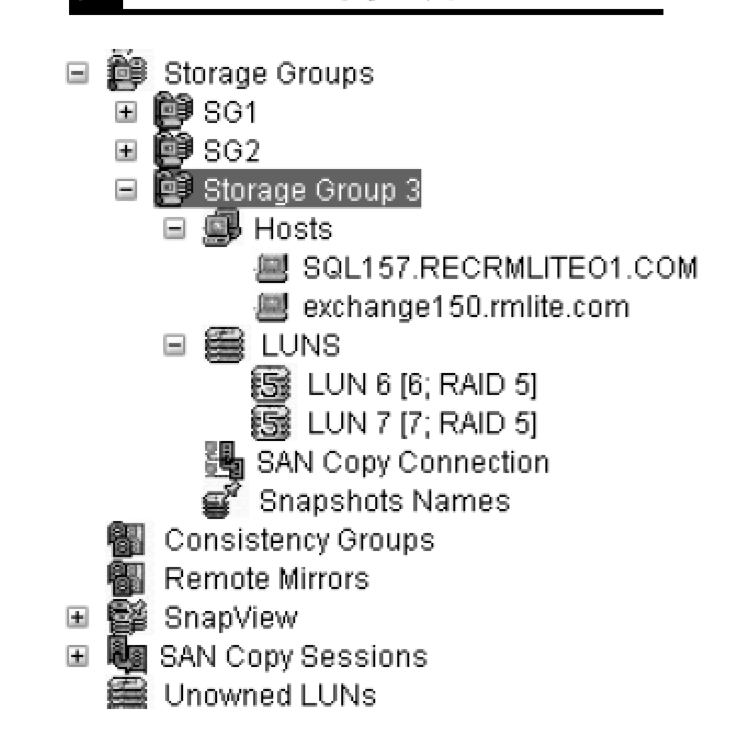


图9.94　创建好的 Storage Group

各种磁盘阵列的配置都是大同小异的，总的来说主要有以下三步。

1】 配置 RAID 组。

2】 在 RAID 组上划分 LUN。

3】 将 LUN 映射给相应的主机使用。

9.6　小　　结

磁盘阵列的前端和后端用 FC 网络作为通路取代原来的并行 SCSI 通路技术，获得了极大的成功！如果说只将主机通道替换成 FC 通路只能称为半网络的话，那么将一个个磁盘作为 FC 网络上的节点对待，可就是彻彻底底的网络化存储系统了。这也自然阐释了“网络存储”和“存储网络”的概念。

> **提示** 此时，我们再来看本书第 2 章中所描述的那个“网中有网”的模型，可以看到，整个系统，所有通路，都可以说是网络化的了，只不过 CPU 内存总线和主机 IO 总线应该算是半网络化，只是一个总线，上面没有网络化所特有的“协议”和“开销”、包交换等词汇。

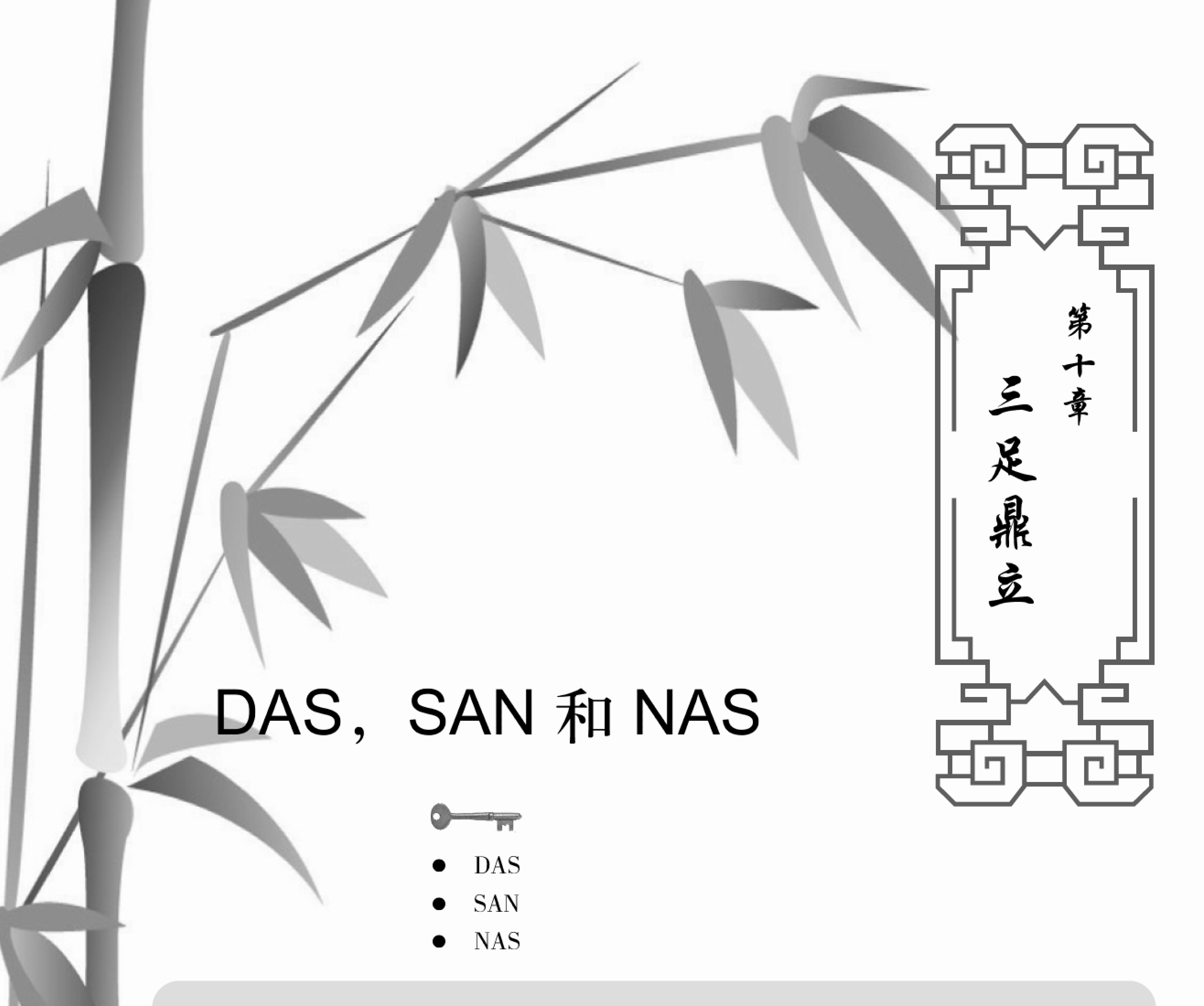
目前，NGIO(Next Generation IO)已经提上日程。这种架构就是将主机 IO 总线甚至内存总线都用交换式网络来连接。可想而知，这个交换网络速率肯定很高，稳定性和可扩展性也很强。

到了 Next Generation 时代，内存、CPU 和各种 IO 设备可以在地理上相隔很远，甚至可以通过网络共享。内存可以不仅仅被一个 CPU 使用，多个 CPU、多个内存、多个 IO 控制器之间都是点对点交换式互联通信。这是一个很有吸引力的课题，就像存储网络化一样。对于存储网络化，位于 A 地的主机可以识别并适用远在相隔千里的 B 地一台盘阵上的 LUN。那么对于网络化的系统总线，A 地的 CPU 可以访问位于 B 地的内存阵列或者位于 B 地的某个 IO 控制器，它们之间都通过网络相连。

有人将上面描述的架构，称为 **System Area Network**，即 **SAN(系统区域网络)**。有意思的是，它和 Storage Area Network 的简称同名。

天道酬　　(敖青云著)
上穷碧落下黄泉，
为索原理细探源。
若非前贤勤耕作，
何来经典广流传？

读 书 笔 记



DAS，SAN 和 NAS

- DAS
- SAN
- NAS

FC 已经成功地将传统的磁盘阵列改造成了彻底网络化传输的磁盘阵列，不仅从盘阵到主机的通路成了网络化，就连盘阵后端控制器到磁盘的连接也被彻底网络化了。尤其对盘阵后端的改革更是一个惊人的创举！

盘阵后端的网络化使可接入磁盘节点数大大增加，可扩展性大大增强。一时间各个厂家纷纷制造出自己的盘阵，由于后端接入容量增加，这些盘阵不是几个磁盘箱就能放下的了，它们动辄就要占用几个大机柜。机房中占地最大的往往就是存储设备，而不是主机设备。

存储区域网络(Storage Area Network，SAN)这个概念，直到 FC 革命成功之后，其意义才真正的体现出来，存储才真正走向了网络化。在广义中，各种存储架构都可以称为 SAN，因为就算直接连到主板上的 IDE 通道也可以连接两块磁盘。从这种意义上说，它就是一个 2 节点网络。

10.1 NAS 也疯狂

武当及张真人所创。想当年，张真人就是在武当创立了卷管理和文件系统的伟大理论！如今，张真人已经逝去多年。然而，他的理论却被广泛的应用着，而且深入人心。不过，没人会追究起到底是谁创立了这些理论，因为它太广泛了，广泛地以至于没有人去理会它了。

10.1.1 另辟蹊径——乱弹 NAS 的起家

武当也跟上了时代的变化，不但本身的体制从公有制改成了股份制，而且董事会还决定将武当仓库全面对外开放，利用一切可以利用的手段盈利。刨去货币贬值的因素，收费比张真人时代贵出了好几倍。利润大部分属于武当董事长瓜董，剩下的除了给员工开点工资外，全部用来扩容仓库和加强仓库建设，以获取更大的利润。

1. 武当仓库简要介绍

武当仓库是一个历史悠久、源远流长的大型仓库，其创始人是张真人。起初只是为了满足武当本派存放货物使用，后来对外开放。仓库分为八个库区，每个库区中又有不计其数的房间。整个仓库配备了两名仓库管理员，各自管理四个库区。一旦某个管理员请假不能到岗，另外一位管理员就要暂时管理全部八个库区。仓库共有东西两道大门，各由一位库管员把守。仓库地理位置优越，其前方就是一个立交桥大枢纽系统，当地路政部门给予武当极其优越的条件，专门为武当仓库的两道大门修了能直通大枢纽的高速路。

武当仓库当前的运作模式如图 10.1 和图 10.2 所示。

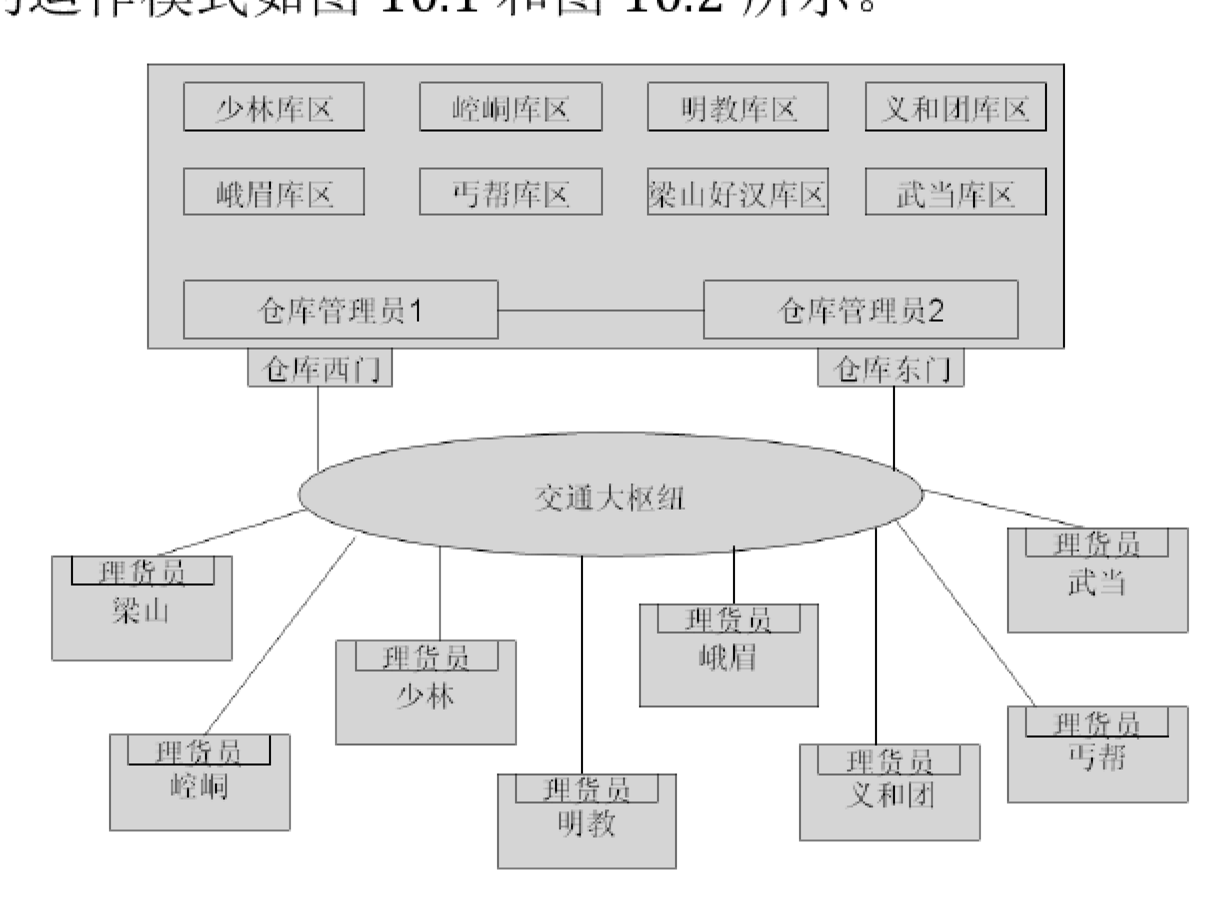


图10.1 武当仓库当前运作模式

自从武当仓库宣布对外全面开放后，短短的一天内，八个库区全部被预售一空。买家当然都是赫赫有名的大门派，因为也只有他们才出得起高价。

由于体改的时候，当年被张真人看好的微软道长被瓜董打发回家了，所以现在武当仓库只有两位根据提货单和入库单进行取货、存货的库管员。

当前武当提货单/存货单格式如图 10.3 所示。

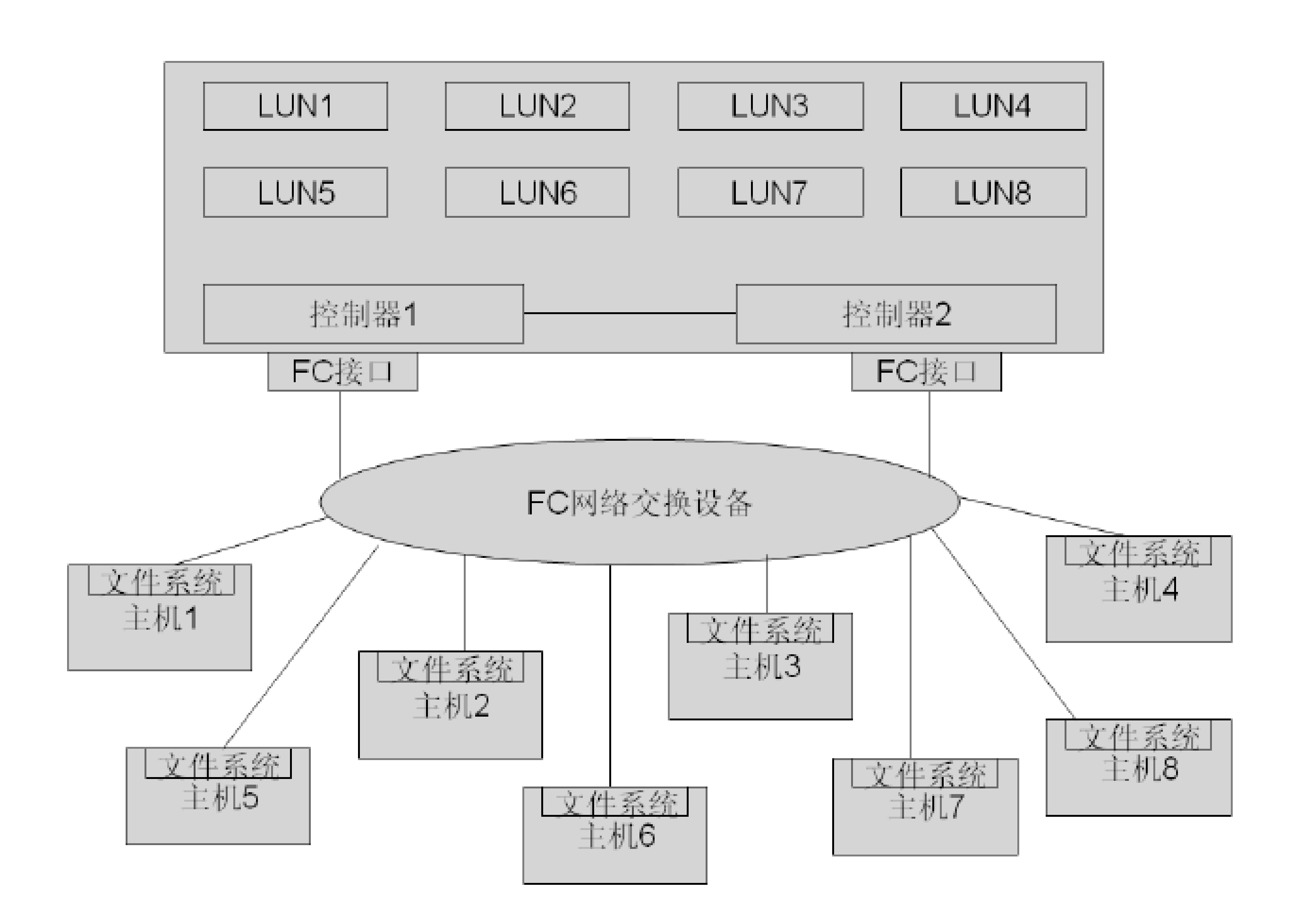


图10.2 当前后端存储网络架构

武当仓库货物存取单　年　月　日

客户名称：________　存货 □　取货 □

库区编号：________

要存取的起始房间编号：________

存取此后的连续房间的数量：________

第一联

客户签字盖章：

仓库恒久远，武当永流传。今年过节不存货，存货就存武当库！　江湖存储管理局监制

图10.3 当前武当仓库货单

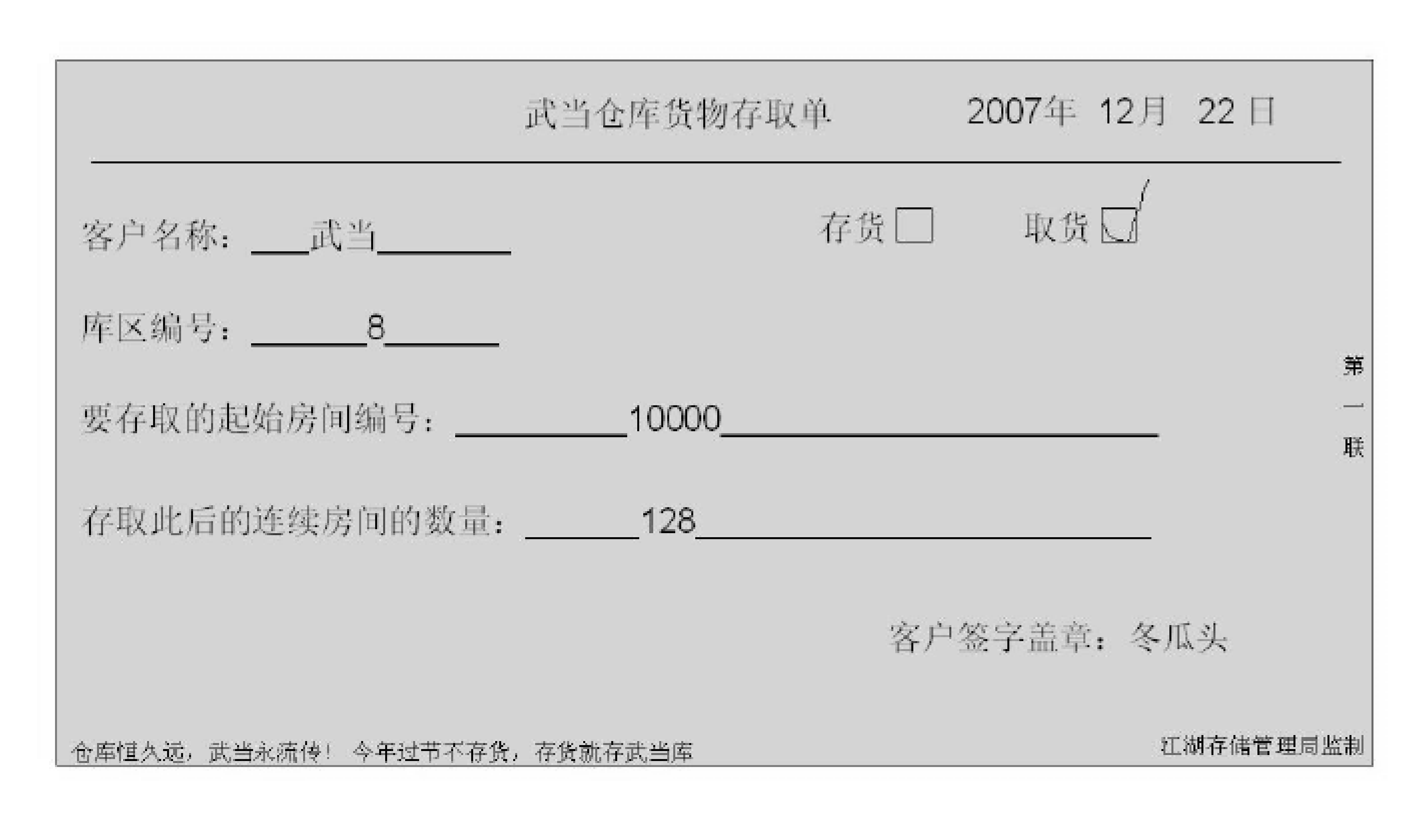
每个门派都通过交通枢纽来向武当仓库存取货物。武当存取货物的单据上没有对货物属性的描述，仓库管理员只管根据货物存取单据上描述的房间号段来将对应的货物取出或者存入，而不管这些货物是什么，有多少。

所以，每个门派要有一个理货员，这个理货员知道什么货应该去哪些房间提取，以及有多少个空闲房间。他自己保留了一个房间使用情况图，每当本门派需要提取什么货物，理货员就根据这个对应图计算出货物对应的存储房间，然后填写武当仓库货物存取单交给仓库管理员，管理员将对应房间中的所有货物交给理货员，理货员在将货物整理好后交给本门派使用。存货的过程也类似，理货员记录好货物要存入的房间，然后填写货物存取单，将存取单和货物交给仓库管理员。管理员根据房间的号码将货物依次放入对应房间。值得说明的是，货物存取单上的房间必须是连续的，不允许断开存放。连续的房间数量最大是 128 个，超过 128 个房间的货物，就需要填写多张货物存取单了，如图 10.4 所示。

武当瓜董天天研究着怎么从现有的资源中，榨取最后一滴利润。他经过调查，发现各门派都养着一个理货员。瓜董心想其他门派一定也很头疼，能少养人就少养人。这天晚上，反复琢磨，他终于想到一个绝招。由于各门派目前都养着一个理货员，他们要付劳务费。如果用武当的人来充当理货员，卖服务给各门派。而每个门派都会付一份劳务费给我，而我只需要付一份工资给理货员就可以了。真是一本万利啊！

另外，由于江湖政府换届时，瓜董没有搞好关系，使武当每月需要向江湖政府缴纳高

额的高速路使用费，这让一向以节省成本著称的瓜董苦不堪言。瓜董决定抛弃高速公路，使用普通公路。


武当仓库货物存取单　　2007年 12月 22 日

客户名称：武当　　存货☐　　取货☑

库区编号：8

要存取的起始房间编号：10000

存取此后的连续房间的数量：128

第一联

客户签字盖章：冬瓜头

仓库恒久远，武当永流传！ 今年过节不存货，存货就存武当库　　江湖存储管理局监制

图10.4　填好的货单

值得称赞的是，瓜董不是个爱面子的人，他为了节省成本，可以不惜一切代价。他亲自把微软道长请回了武当，让他担任仓库理货大总管职务。随后，瓜董联系了各门各派掌门人，让他们把理货的工作外包给武当做，劳务费比原来有所优惠。各门派都同意了。

微软道长手下可以有多个分管。各个门派使用的货物记录方式并不相同，大部分门派使用的是微软道长所创立的 NTFS 记录方式，而有些则使用的是少林雷牛大师所创的 EXT 格式。既然要将货物记录服务全部外包给武当，那么微软道长无疑要将所有这些门派使用的记录方式都实现，从而为每个门派服务。微软道长遂发布告示，广招天下贤士来任职分管职务，每个分管管理其各自的货物记录方式。图 10.5 和图 10.6 为改制之后的仓库管理模式。

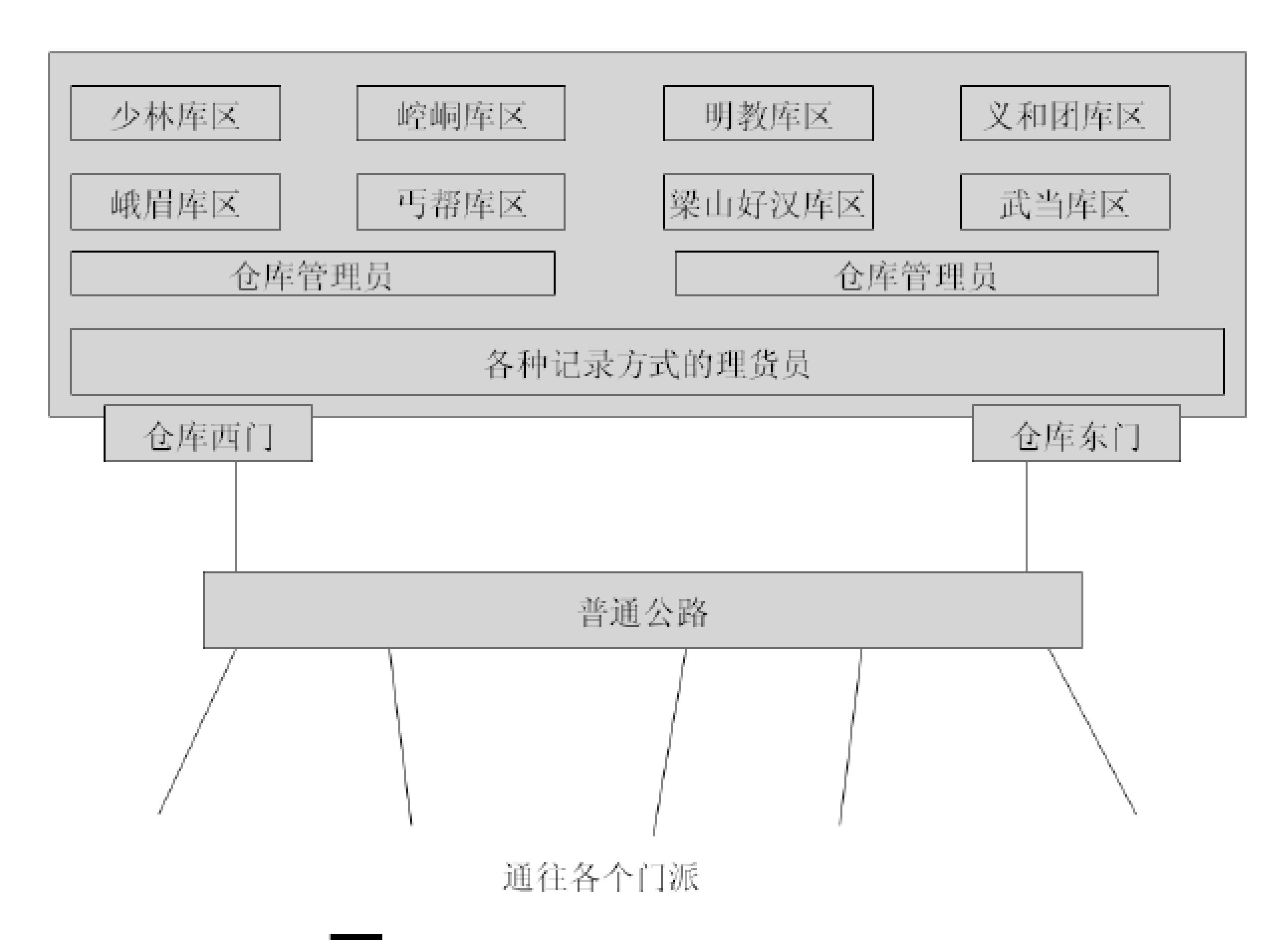


图10.5　改制之后的仓库架构

2. 改制之后的仓库架构

武当仓库经过这样的改造之后，功能更加强大了。惟一不足的是为了节省成本将原本高成本的高速公路替换成了普通公路。

原来的单据显然不适合仓库当前的运作模式了，仓库的货物存取单也改版了，如图 10.7 所示。

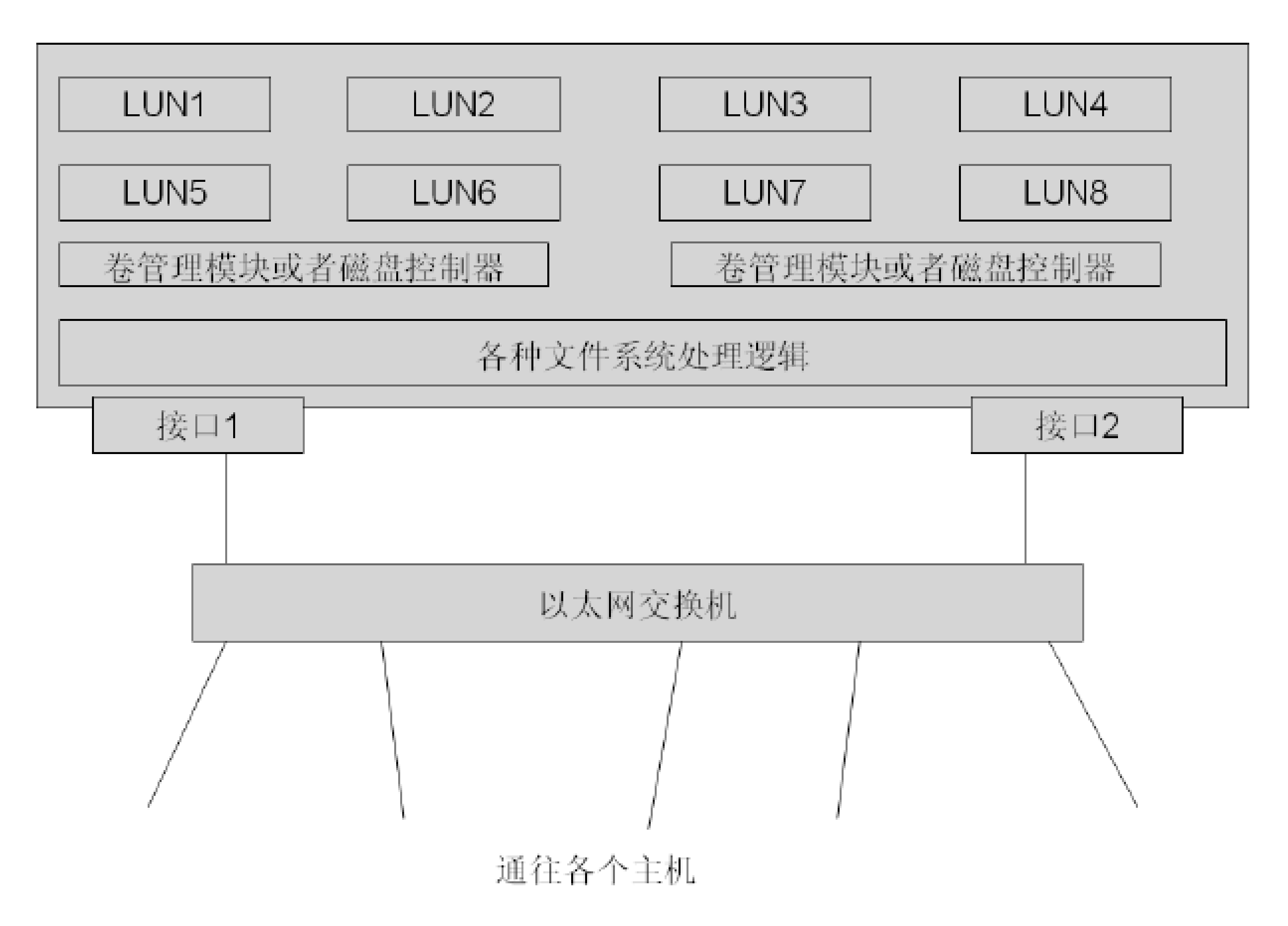


图10.6 改制之后的后端存储网络架构

武当仓库货物存取单（V2.0） 2007年 12月 22 日

客户名称：___武当_______ 存货 □ 取货 ☑

需要存取的货物类别：______蔬菜类______

要存取的货物名称：_________冬瓜______________________

要存取的货物质量：______128___千克

客户签字盖章：冬瓜头

第一联

仓库恒久远，武当永流传！ 今年过节不存货，存货就存武当库 江湖存储管理局监制

图10.7 改制之后的货单

从武当仓库整改之后的新单据中可以发现，现在各个门派只要告诉武当仓库的理货员需要什么货物及多少数量就可以了，完全不用记录这些货物实际放到了哪个房间，在哪里及怎么去这些内容。计算这些复杂的对应关系的工作完全移交给了武当仓库理货员来做。理货员计算好之后，生成提货/存货单，交给库管员，直接去对应房间提/存货物。理货员与库管员之间的交互，速度快了很多，因为完全是在仓库内部进行通信了，不需要跨越缓慢的公路交通系统。

经过实践，瓜董的这套做法还真取得了显著的成效，各门派无须雇用理货员，无须支付昂贵的高速费，节省了成本。同时，瓜董也发了财。不过惟一不足的地方就是用普通公路进行货运的速度，各门派不太满意。但是相对于成本大大地降低和便捷带来的好处，各门派也只有牺牲速度了。

10.1.2 双管齐下——两种方式访问的后端存储网络

但是有的仓库租用者对这种方案并没有兴趣，他们不但追求速度，而且情愿用自己的理货员，也不相信仓库提供的理货员。这种客户得罪不起，那么就给他单独的政策，还是采用原来的方式使用仓库，和新方式互不影响。在仓库前面开辟了新的道路去接入高速枢纽。这样就满足了两种不同的需求，如图 10.8 和图 10.9 所示。

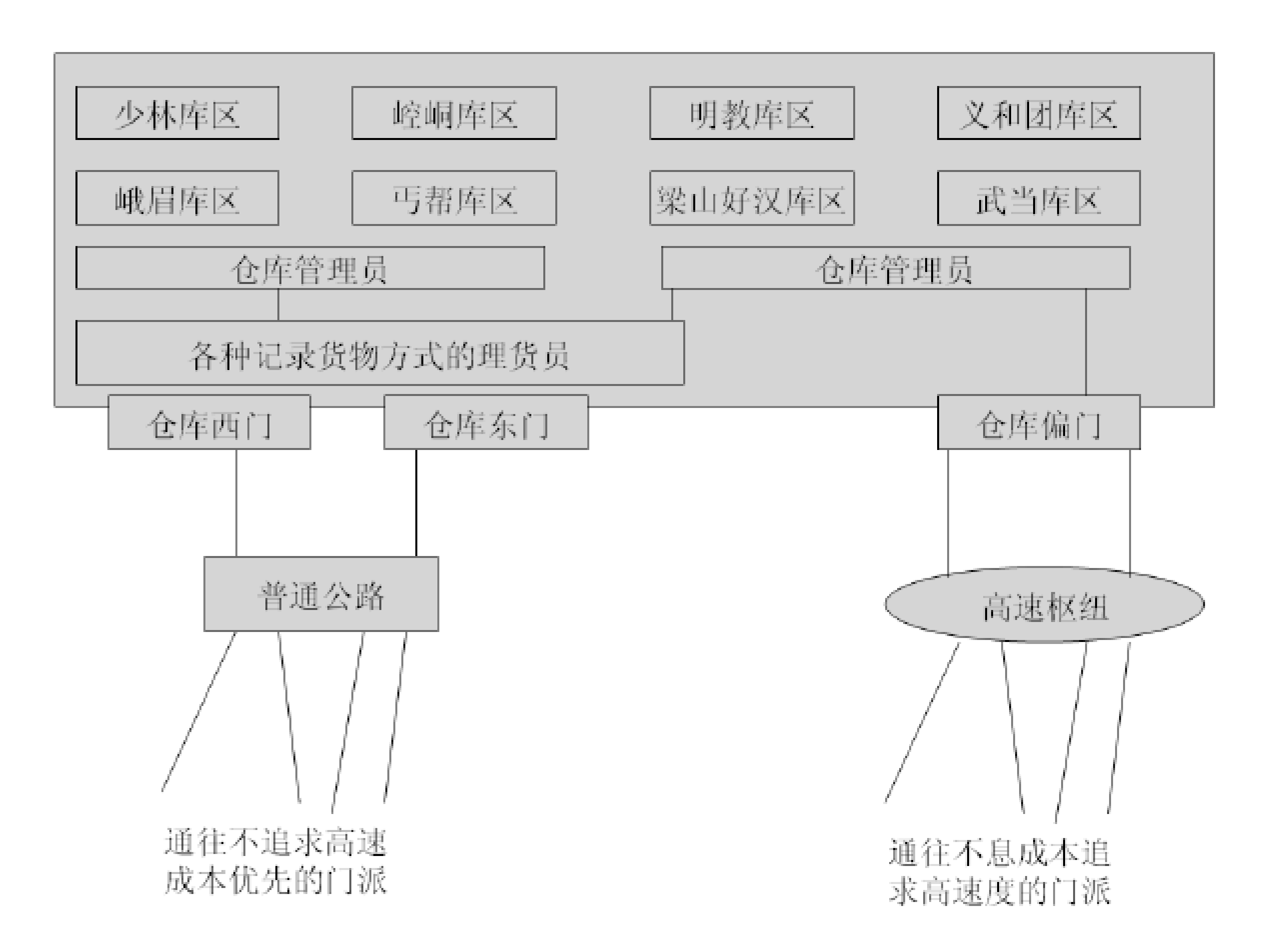


图10.8 旁路传统访问

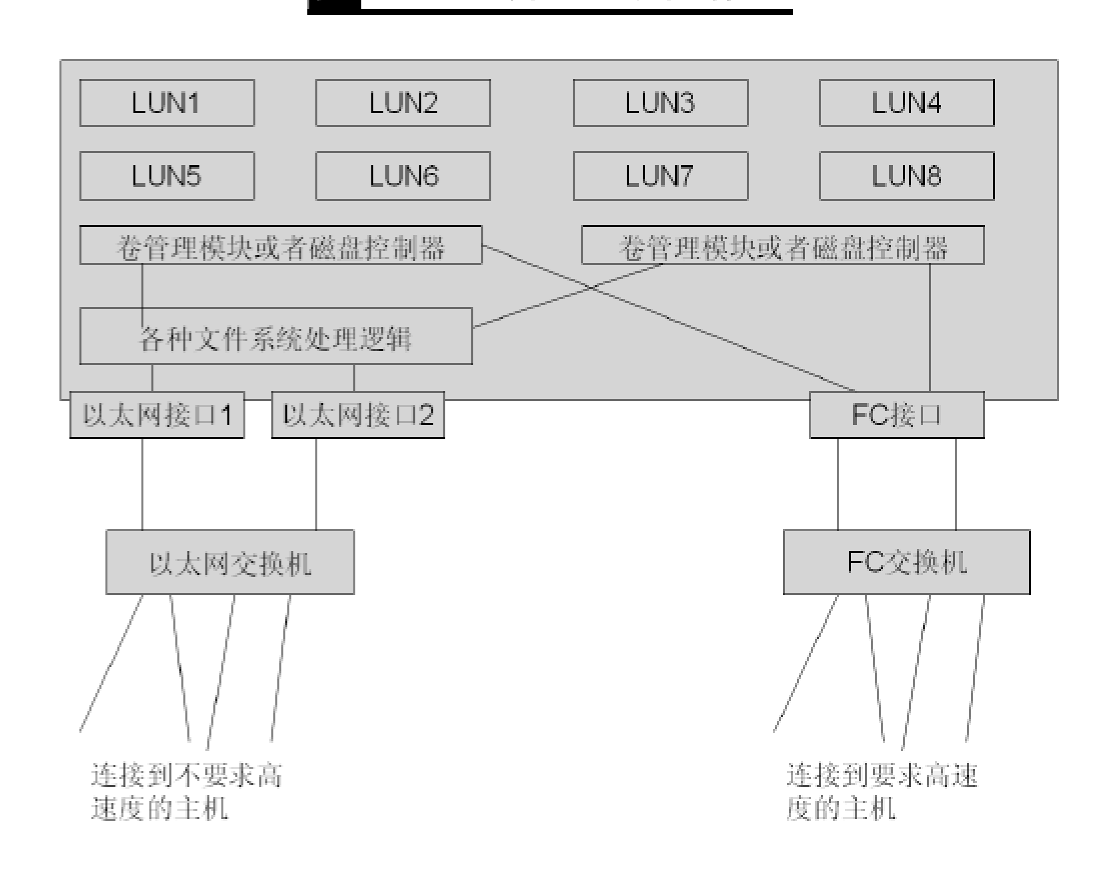


图10.9 两种方式访问的后端存储网络

10.1.3 万物归一——网络文件系统

微软老道在吸取瓜董的思想之后，终于创立了自己的理论。武当的仓库就像一个大的卷，一个大的磁盘阵列。它可以划分出多个 LUN 供多个使用者使用。而每个使用者必须有自己的文件系统，因为这个 LUN 只是一个卷设备，只提供了不计其数的房间存放货物。至于怎么存放货物。需要由使用者自己决定和管理，也就是用文件系统来管理卷，像理货员做货物记录一样。

微软老道把瓜董的思想用在了存储上，他把文件系统的功能从使用者处迁移到了磁盘阵列之上，让磁盘阵列自己管理存储空间。而对外提供统一的用户接口(货物存取单据 v2.0)，使得使用者不用再记录某某文件和卷上扇区或者簇块的对应关系，这个工作统统由盘阵上的集中式文件系统模块处理。使用者只需通过网络告诉这个文件系统需要存取什么文件，长度是多少就可以了。具体存取数据的过程，由集中式文件系统来做，使用者只要等待接受数据就可以了。同样，在存文件的时候，使用者只需告诉文件系统要存那些数据，提供一些文件名、长度、哪个目录下等信息就可以了，至于文件存到卷的哪些空余扇区完全由盘阵上的文件系统逻辑来处理，使用者不必关心。

位于盘阵上的集中式文件系统得益于包交换网络，可以同时处理多个使用者的请求。它可以给每个使用者提供各自的文件夹目录，并且可以为这些目录限定允许存放的最大数据量。总之，文件系统可以实现的任何功能都可以在盘阵上实现。

网络文件系统

使用者如何与盘阵上的集中式文件系统进行交互呢？当然是通过网络来传递数据。由于一直以来的习惯，以太网加 TCP/IP 成为了首选的网络方式。除了底层传输网络，还必须定义上层的应用逻辑。

针对上层逻辑，微软定义了自己的一套规范，叫做 CIFS(Common Internet File System)，意思是 Internet 范围的 FS。Linux 和 UNIX 系统使用了另一种方式，称为 NFS(Network File System)，这些上层协议都是利用 TCP/IP 协议进行传输的。

以上描述的模型统称为“网络文件系统”。这种文件系统逻辑不是在本地运行，而是在网络上的其他节点运行，使用者通过外部网络将读写文件的信息传递给运行在远端的文件系统，也就是调用远程的文件系统模块，而不是在本地内存中调用文件系统的 API 来进行。所以网络文件系统又叫做远程调用式文件系统，也就是 RPC FS(Remote Procedure Call File System)。

相对于 SAN 来说，这种网络文件系统不仅仅磁盘或卷在远程节点上，连文件系统功能也搬运到了远程节点上。本地文件系统可以直接通过主板上的导线访问内存来调用其功能。而网络文件系统只能通过网络适配器上连接的网线而不是主板上的导线来访问远端的文件系统功能。

网络文件系统在网络上传递的是些什么内容呢？下面我们用抓包的方式分析一下。

1) CIFS 协议网络包分析

在某个用 CIFS 访问方式的目录下新建一个文本文档，然后将它删除。此过程中抓取网络流量。在 CIFS 方式下，仅仅上述两个动作，就引发了网络上数百个包的流量(如图 10.10 所示)，可见 CIFS 是一个开销非常大的 NAS 协议。这里就不一一分析每个包了。

No.	Time	Source	Destination	Protocol	Info
251	10.364194	10.128.132.175	10.128.128.194	SMB	NT Trans Response, NT NOTIFY
252	10.364479	10.128.128.194	10.128.132.175	SMB	NT Trans Request, NT NOTIFY, FID: 0x0304
254	10.857237	10.128.128.194	10.128.132.175	SMB	NT Create AndX Request, Path:
255	10.857275	10.128.132.175	10.128.128.194	SMB	NT Create AndX Response, FID: 0x035d, FID: 0x035d
256	10.857561	10.128.128.194	10.128.132.175	SMB	NT Trans Request, NT IOCTL NETWORK_FILE_SYSTEM Function:0x0019, FI
257	10.857590	10.128.132.175	10.128.128.194	SMB	NT Trans Response, FID: 0x035d, NT IOCTL
258	10.857876	10.128.128.194	10.128.132.175	SMB	NT Trans Request, NT IOCTL NETWORK_FILE_SYSTEM Function:0x0019, FI
259	10.857906	10.128.132.175	10.128.128.194	SMB	NT Trans Response, FID: 0x035d, NT IOCTL
260	10.858286	10.128.128.194	10.128.132.175	SMB	Trans2 Request, QUERY_PATH_INFO, Query File Basic Info, Path:
261	10.858311	10.128.132.175	10.128.128.194	SMB	Trans2 Response, QUERY_PATH_INFO
262	10.858870	10.128.128.194	10.128.132.175	SMB	NT Create AndX Request, Path: \
263	10.858893	10.128.132.175	10.128.128.194	SMB	NT Create AndX Response, FID: 0x035e, FID: 0x035e
264	10.859086	10.128.128.194	10.128.132.175	SMB	NT Trans Request, NT IOCTL FILE_SYSTEM Function:0x002a, FID: 0x03
265	10.859094	10.128.132.175	10.128.128.194	SMB	NT Trans Response, FID: 0x035e, NT IOCTL, Error: STATUS_INVALID_P
266	10.859380	10.128.128.194	10.128.132.175	SMB	Close Request, FID: 0x035e
267	10.859398	10.128.132.175	10.128.128.194	SMB	Close Response, FID: 0x035e
268	10.859775	10.128.128.194	10.128.132.175	SMB	Close Request, FID: 0x035d
269	10.859787	10.128.132.175	10.128.128.194	SMB	Close Response, FID: 0x035d
273	11.853783	10.128.128.194	10.128.132.175	SMB	Trans2 Request, FIND_FIRST2, Pattern: *
274	11.853911	10.128.132.175	10.128.128.194	SMB	Trans2 Response, FIND_FIRST2, Files: ~snapshot etc home 724_sysfil
290	12.854569	10.128.128.194	10.128.132.175	SMB	NT Create AndX Request, Path: \New Text Document.txt
291	12.854630	10.128.132.175	10.128.128.194	SMB	NT Create AndX Response, FID: 0x0000, Error: STATUS_OBJECT_NAME_NO
292	12.854915	10.128.128.194	10.128.132.175	SMB	Trans2 Request, QUERY_PATH_INFO, Query File Basic Info, Path: \New
293	12.854935	10.128.132.175	10.128.128.194	SMB	Trans2 Response, QUERY_PATH_INFO, Error: STATUS_OBJECT_NAME_NOT_FO
294	12.855312	10.128.128.194	10.128.132.175	SMB	Trans2 Request, QUERY_PATH_INFO, Query File Basic Info, Path: \New
295	12.855327	10.128.132.175	10.128.128.194	SMB	Trans2 Response, QUERY_PATH_INFO, Error: STATUS_OBJECT_NAME_NOT_FO
296	12.855613	10.128.128.194	10.128.132.175	SMB	Trans2 Request, QUERY_PATH_INFO, Query File Basic Info, Path: \New
297	12.855627	10.128.132.175	10.128.128.194	SMB	Trans2 Response, QUERY_PATH_INFO, Error: STATUS_OBJECT_NAME_NOT_FO
298	12.855912	10.128.128.194	10.128.132.175	SMB	Trans2 Request, QUERY_PATH_INFO, Query File Basic Info, Path: \New
299	12.855926	10.128.132.175	10.128.128.194	SMB	Trans2 Response, QUERY_PATH_INFO, Error: STATUS_OBJECT_NAME_NOT_FO
300	12.856486	10.128.128.194	10.128.132.175	SMB	NT Create AndX Request, Path: \
301	12.856519	10.128.132.175	10.128.128.194	SMB	NT Create AndX Response, FID: 0x0361, FID: 0x0361
302	12.856713	10.128.128.194	10.128.132.175	SMB	NT Trans Request, NT IOCTL FILE_SYSTEM Function:0x002a, FID: 0x03
303	12.856721	10.128.132.175	10.128.128.194	SMB	NT Trans Response, FID: 0x0361, NT IOCTL, Error: STATUS_INVALID_P
304	12.856915	10.128.128.194	10.128.132.175	SMB	Close Request, FID: 0x0361
305	12.856935	10.128.132.175	10.128.128.194	SMB	Close Response, FID: 0x0361

图10.10 CIFS 协议交互的数据包

2) NFS 协议网络包分析

在一台 Linux 客户端上使用 NFSv3 来 Mount 一台 NFS 服务器上的某个目录到本地的

/mnt 目录下。进入这个目录，然后用“touch a”命令来创建一个名为“a”的文件，然后执行“vi a”，进入编辑模式，不做任何修改，退出，之后用“rm a”命令来删除这个文件。其间抓取网络上的流量。


10.128.132.45 是 NFS 客户端的 IP 地址，10.128.132.175 是 NFS 服务器端的 IP 地址。分析结果已经去除了不必要的 TCP 包以及 TCP_ACK 等包。

图 10.11 显示的是这个过程中网络上双方所交互的主要数据包。

No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.128.132.45	10.128.132.175	NFS	V3 LOOKUP Call, DH:0x98f8d6bb/a
2	0.000052	10.128.132.175	10.128.132.45	NFS	V3 LOOKUP Reply (Call In 1) Error:NFS3ERR_NOENT
4	0.000247	10.128.132.45	10.128.132.175	NFS	V3 CREATE Call, DH:0x98f8d6bb/a Mode:UNCHECKED
5	0.000315	10.128.132.175	10.128.132.45	NFS	V3 CREATE Reply (Call In 4)
6	0.000511	10.128.132.45	10.128.132.175	NFS	V3 GETATTR Call, FH:0xf03ce91c
7	0.000523	10.128.132.175	10.128.132.45	NFS	V3 GETATTR Reply (Call In 6) Regular File mode:0644 uid:0 gid:0
8	0.000718	10.128.132.45	10.128.132.175	NFS	V3 GETATTR Call, FH:0x98f8d6bb
9	0.000728	10.128.132.175	10.128.132.45	NFS	V3 GETATTR Reply (Call In 8) Directory mode:0777 uid:0 gid:0
10	0.000922	10.128.132.45	10.128.132.175	NFS	V3 LOOKUP Call, DH:0x98f8d6bb/a
11	0.000934	10.128.132.175	10.128.132.45	NFS	V3 LOOKUP Reply (Call In 10), FH:0xf03ce91c
12	0.001129	10.128.132.45	10.128.132.175	NFS	V3 SETATTR Call, FH:0xf03ce91c
13	0.001157	10.128.132.175	10.128.132.45	NFS	V3 SETATTR Reply (Call In 12)
15	5.648643	10.128.132.45	10.128.132.175	NFS	V3 ACCESS Call, FH:0x98f8d6bb
16	5.648706	10.128.132.175	10.128.132.45	NFS	V3 ACCESS Reply (Call In 15)
18	5.648902	10.128.132.45	10.128.132.175	NFS	V3 GETATTR Call, FH:0xf03ce91c
19	5.648915	10.128.132.175	10.128.132.45	NFS	V3 GETATTR Reply (Call In 18) Regular File mode:0644 uid:0 gid:0
20	5.649109	10.128.132.45	10.128.132.175	NFS	V3 ACCESS Call, FH:0xf03ce91c
21	5.649120	10.128.132.175	10.128.132.45	NFS	V3 ACCESS Reply (Call In 20)
23	7.335934	10.128.132.45	10.128.132.175	NFS	V3 REMOVE Call, DH:0x98f8d6bb/a
24	7.336026	10.128.132.175	10.128.132.45	NFS	V3 REMOVE Reply (Call In 23)

图10.11　NFS 方式下网络上的数据包

我们看到，NFS 协议的开销远远小于 CIFS 协议。完成相似动作，NFS 只需要交互十几个包即可。下面来分析每个包的作用。

1】 Frame1(如图 10.12 所示)：客户端在创建文件之前，首先做了一次 lookup 操作，来查找当前目录中是否已经有同名文件，如果有，则拒绝创建。图中的 DH 表示 Directory Handle，是一个 32 字节长的字段，这个值用来指代目录名称，在第一次访问某个目录时，NFS 服务端会动态分配这个值，将其通知给客户端，随后的访问请求中，客户端将使用这个值而不是目录名称来向 NFS 服务端发起针对这个目录的请求。


图中 DH 值的 hash 值为 0x98f8d6bb，为了表示方便，抓包软件将其 hash 成一个 4 字节的值，这个 hash 值并不是存在于网络包中的。本例中，/mnt 目录被指定的 DH 值的 hash 值就是 0x98f8d6bb。

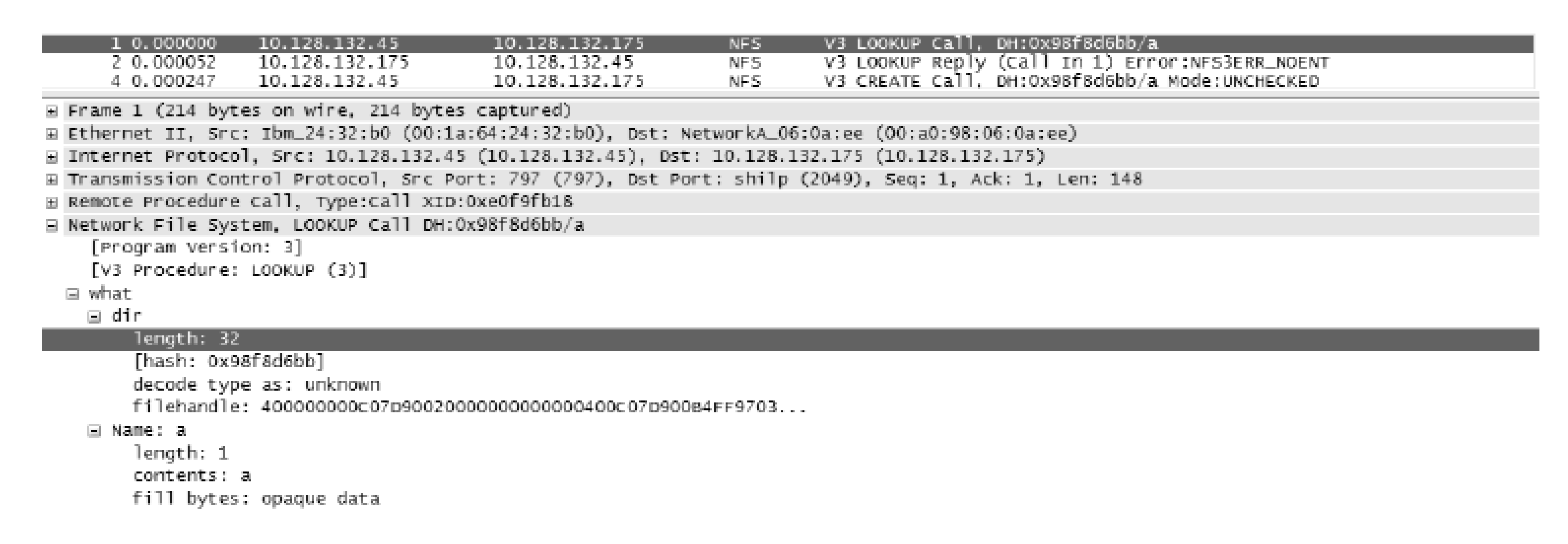

```
1 0.000000   10.128.132.45    10.128.132.175   NFS   V3 LOOKUP Call, DH:0x98f8d6bb/a
2 0.000052   10.128.132.175   10.128.132.45    NFS   V3 LOOKUP Reply (Call In 1) Error:NFS3ERR_NOENT
4 0.000247   10.128.132.45    10.128.132.175   NFS   V3 CREATE Call, DH:0x98f8d6bb/a Mode:UNCHECKED
⊞ Frame 1 (214 bytes on wire, 214 bytes captured)
⊞ Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
⊞ Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
⊞ Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 1, Ack: 1, Len: 148
⊞ Remote Procedure Call, Type:Call XID:0xe0f9fb18
⊟ Network File System, LOOKUP Call DH:0x98f8d6bb/a
    [Program Version: 3]
    [V3 Procedure: LOOKUP (3)]
  ⊟ what
    ⊟ dir
        length: 32
        [hash: 0x98f8d6bb]
        decode type as: unknown
        filehandle: 400000000c07D90020000000000000400c07D90084FF9703...
    ⊟ Name: a
        length: 1
        contents: a
        fill bytes: opaque data
```

图10.12　客户端的 Lookup 请求

2】 Frame2(如图 10.13 所示)：为 NFS 服务端对 Frame1 的回应。通过“ERR_NOENT”可以判断出当前目录并没有名为“a”的文件。

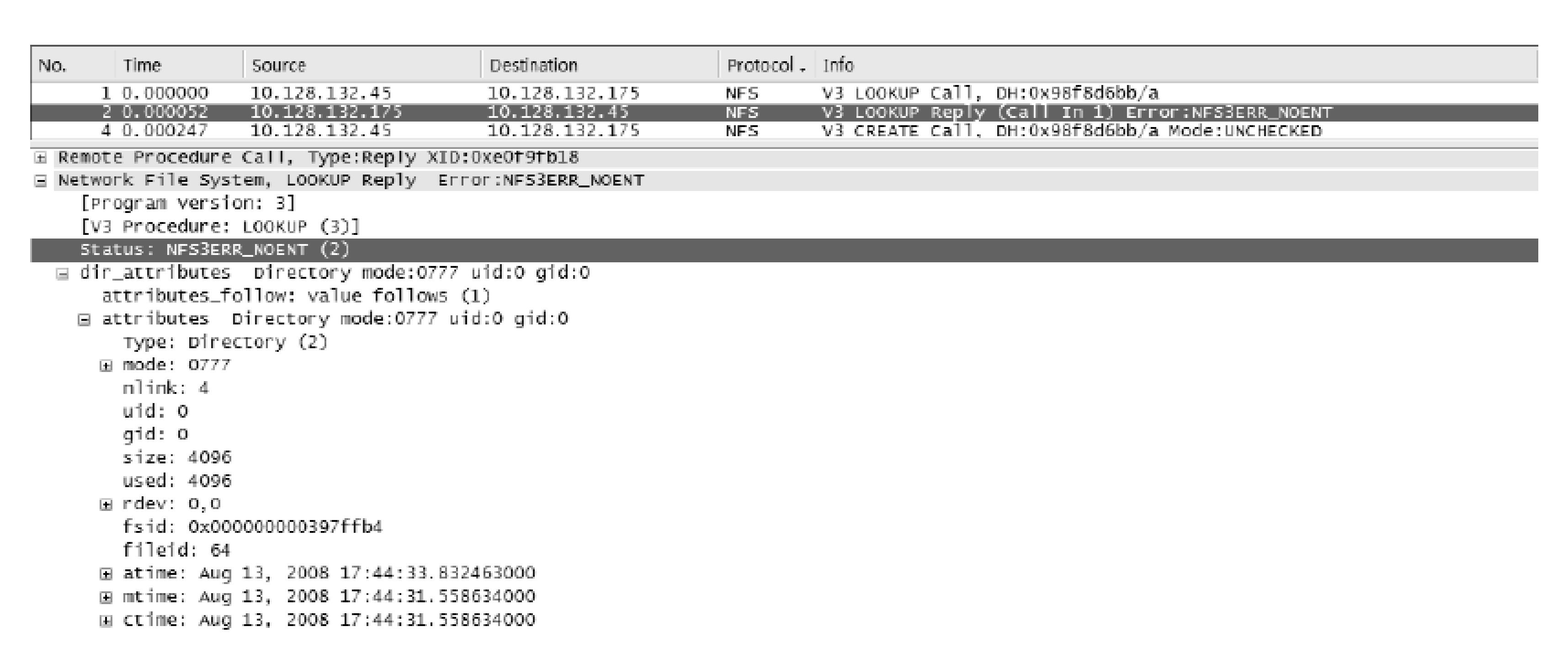


图10.13 Lookup 请求的回应

3】 Frame4(如图 10.14 所示)：客户端随即发起了“Create Call”，创建“a”文件。

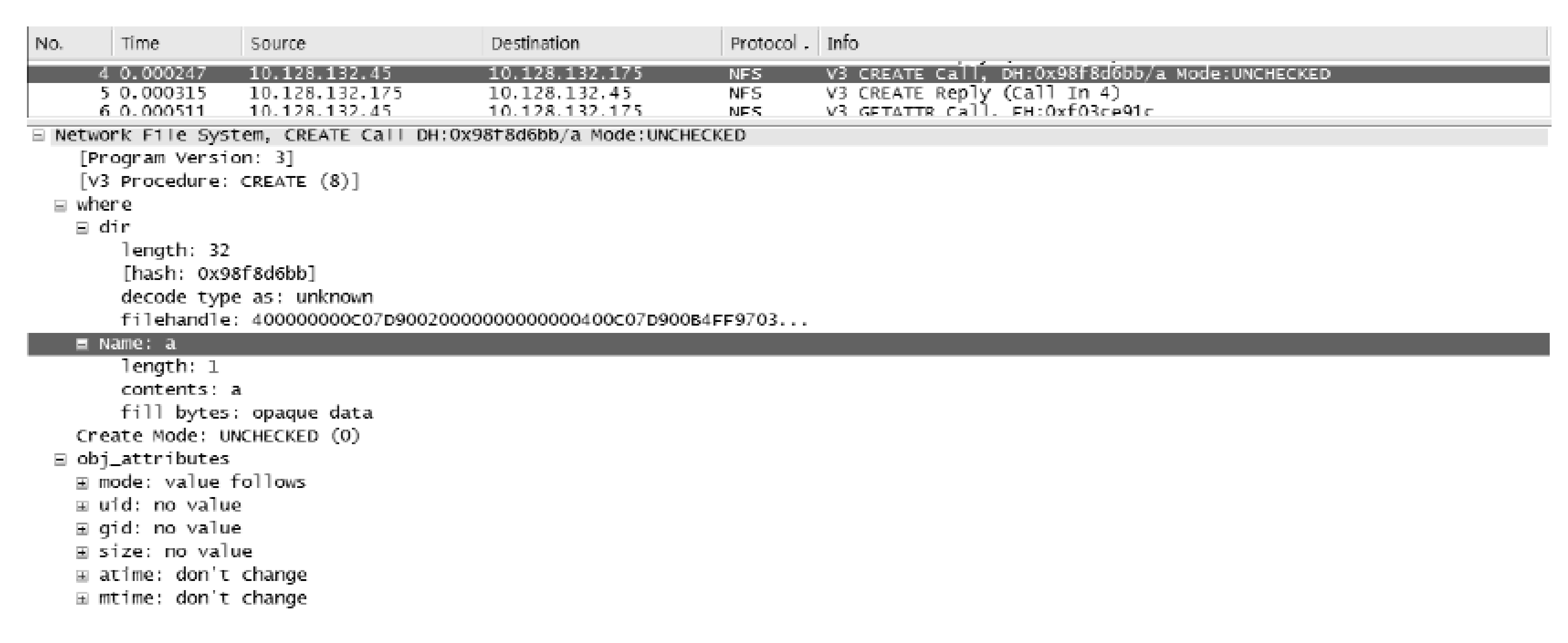


图10.14 客户端的 Create 请求

4】 Frame5(如图 10.15 所示)：NFS 服务端对客户端 Frame4 的回应，创建成功，服务端返回 File Handle 的 hash 值为：0xf03ce91c，FH(File Handle)与 DH 一样，在数据包中实际上也为一个 32 字节长的字段，为了表示方便，抓包软件将其 hash 成一个 4 字节的值。随后的交互中客户端不会用文件名来向服务端请求操作，而全部用这个 File Handle 来指代。

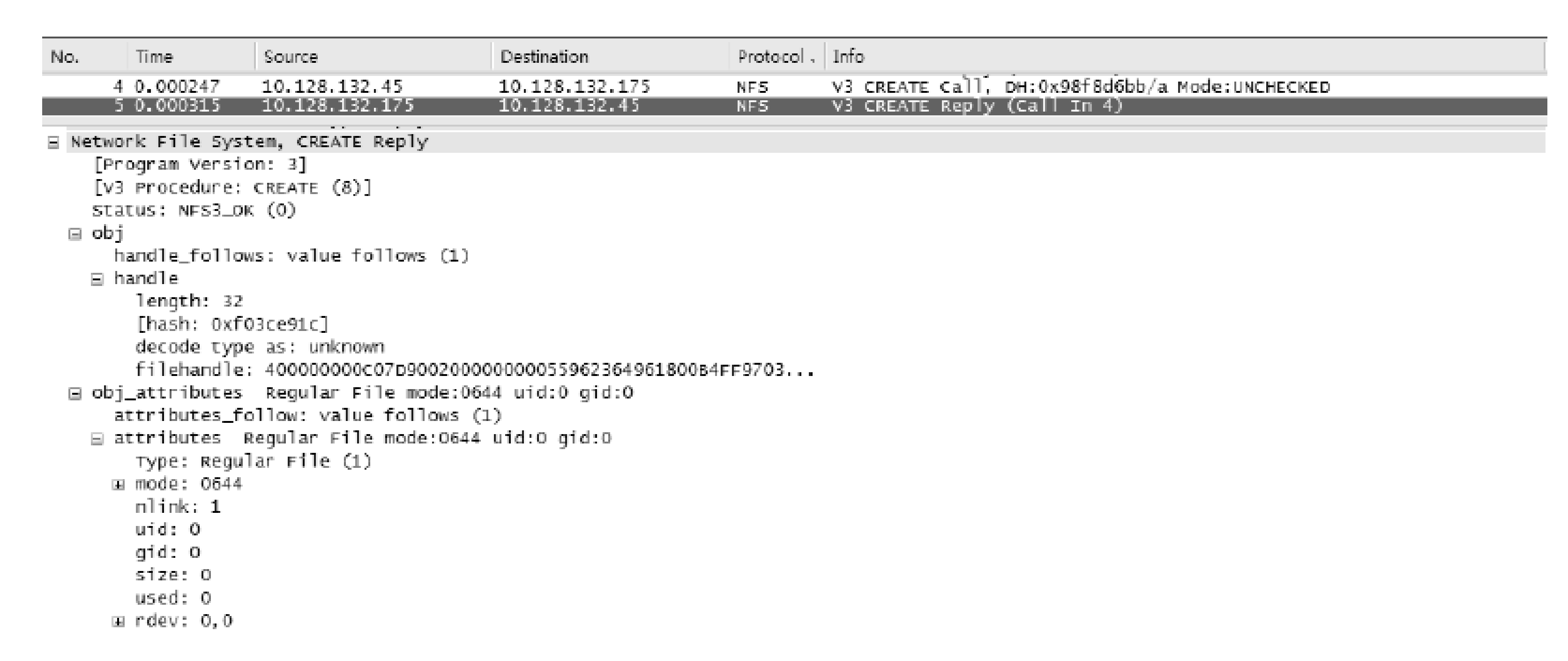


图10.15 Create 请求的回应

5】 Frame6(如图 10.16 所示)：文件“a”创建成功之后，出于保险起见，应用程序一般都会紧接着查询一下文件属性，顺便确认文件是否创建成功。“GetAttr Call”就是用来查询文件属性用的一种 RPC call。图中可以看到对应 FH 值为 0xf03ce91c，NFS 服务端收到这个值就会自动对应成文件“a”。

```
No.   Time       Source          Destination     Protocol  Info
    6 0.000511   10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0xf03ce91c
    7 0.000523   10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 6)  Regular File mode:0644 uid:0 gid:0
⊞ Frame 6 (206 bytes on wire, 206 bytes captured)
⊞ Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
⊞ Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
⊞ Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 329, Ack: 397, Len: 140
⊞ Remote Procedure Call, Type:Call XID:0xe2f9fb18
⊟ Network File System, GETATTR Call FH:0xf03ce91c
    [Program Version: 3]
    [V3 Procedure: GETATTR (1)]
  ⊟ object
      length: 32
      [hash: 0xf03ce91c]
      decode type as: unknown
      filehandle: 400000000C07D900200000000055962364961800B4FF9703...
```

图10.16　客户端的 GetAttr 请求

6】 Frame7(如图 10.17 所示)：NFS 服务端对 Frame6 的回应。包中可以看到文件的 umask 访问权限以及 atime，mtime，ctime 属性。

```
No.   Time       Source          Destination     Protocol  Info
    6 0.000511   10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0xf03ce91c
    7 0.000523   10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 6)  Regular File mode:0644 uid:0 gid:0
⊟ Network File System, GETATTR Reply  Regular File mode:0644 uid:0 gid:0
    [Program Version: 3]
    [V3 Procedure: GETATTR (1)]
    Status: NFS3_OK (0)
  ⊟ obj_attributes  Regular File mode:0644 uid:0 gid:0
      Type: Regular File (1)
    ⊞ mode: 0644
      nlink: 1
      uid: 0
      gid: 0
      size: 0
      used: 0
    ⊞ rdev: 0,0
      fsid: 0x000000000397ffb4
      fileid: 5608995
    ⊞ atime: Aug 13, 2008 17:45:55.545410000
    ⊞ mtime: Aug 13, 2008 17:45:55.545410000
    ⊞ ctime: Aug 13, 2008 17:45:55.545410000
```

图10.17　GetAttr 请求的回应

7】 Frame8(如图 10.18 所示)：紧接着 NFS 客户端发起了一个查询/mnt 目录属性的请求。因为 Handle 的值为 0x98f8d6bb，所以可以判断这个 GetAttr Call 是针对/mnt 目录的。

```
No.   Time       Source          Destination     Protocol  Info
    8 0.000718   10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0x98f8d6bb
    9 0.000728   10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 8)  Directory mode:0777 uid:0 gid:0
⊞ Frame 8 (206 bytes on wire, 206 bytes captured)
⊞ Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
⊞ Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
⊞ Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 469, Ack: 513, Len: 140
⊞ Remote Procedure Call, Type:Call XID:0xe3f9fb18
⊟ Network File System, GETATTR Call FH:0x98f8d6bb
    [Program Version: 3]
    [V3 Procedure: GETATTR (1)]
  ⊟ object
      length: 32
      [hash: 0x98f8d6bb]
      decode type as: unknown
      filehandle: 400000000C07D9002000000000000400C07D900B4FF9703...
```

图10.18　客户端针对/mnt 目录的 GetAttr 请求

8】 Frame9(如图 10.19 所示)：NFS 服务端对 Frame8 的回应。

```
No.   Time       Source          Destination     Protocol  Info
    8 0.000718   10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0x98f8d6bb
    9 0.000728   10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 8)  Directory mode:0777 uid:0 gid:0
⊟ Network File System, GETATTR Reply  Directory mode:0777 uid:0 gid:0
    [Program Version: 3]
    [V3 Procedure: GETATTR (1)]
    Status: NFS3_OK (0)
  ⊟ obj_attributes  Directory mode:0777 uid:0 gid:0
      Type: Directory (2)
    ⊞ mode: 0777
      nlink: 4
      uid: 0
      gid: 0
      size: 4096
      used: 4096
    ⊞ rdev: 0,0
      fsid: 0x000000000397ffb4
      fileid: 64
    ⊞ atime: Aug 13, 2008 17:44:33.832463000
    ⊞ mtime: Aug 13, 2008 17:45:55.545410000
    ⊞ ctime: Aug 13, 2008 17:45:55.545410000
```

图10.19　针对 GetAttr 请求的回应

9】 Frame10(如图 10.20 所示)：NFS 客户端发起一个在/mnt 目录中查找文件“a”的请求。这里由于是查找操作，客户端会假设不知道“a”文件的 FH 值，而只知道/mnt 目录的 DH 值，所以文件名“a”使用的就是 ASCII 码的“a”。

10】 Frame11(如图 10.21 所示)：NFS 服务端根据 Frame10 中请求的回应找到这个文件，FH 值是 0xf03ce91c。

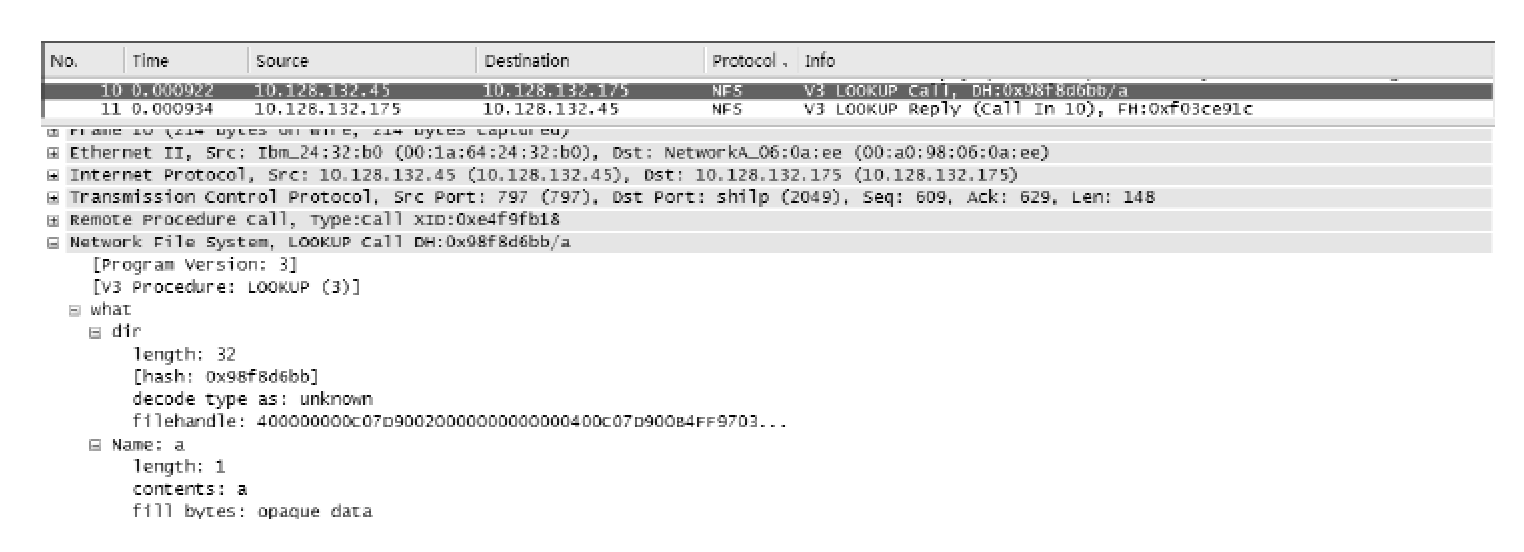


图10.20　客户端的 Lookup 请求

```
No.  Time        Source          Destination     Protocol  Info
  10 0.000922    10.128.132.45   10.128.132.175  NFS       V3 LOOKUP Call, DH:0x98f8d6bb/a
  11 0.000934    10.128.132.175  10.128.132.45   NFS       V3 LOOKUP Reply (Call In 10), FH:0xf03ce91c
⊞ Frame 11 (310 bytes on wire, 310 bytes captured)
⊞ Ethernet II, Src: NetworkA_06:0a:ee (00:a0:98:06:0a:ee), Dst: Ibm_24:32:b0 (00:1a:64:24:32:b0)
⊞ Internet Protocol, Src: 10.128.132.175 (10.128.132.175), Dst: 10.128.132.45 (10.128.132.45)
⊞ Transmission Control Protocol, Src Port: shilp (2049), Dst Port: 797 (797), Seq: 629, Ack: 757, Len: 244
⊞ Remote Procedure Call, Type:Reply XID:0xe4f9fb18
⊟ Network File System, LOOKUP Reply FH:0xf03ce91c
    [Program Version: 3]
    [V3 Procedure: LOOKUP (3)]
    Status: NFS3_OK (0)
  ⊟ object
      length: 32
      [hash: 0xf03ce91c]
      decode type as: unknown
      filehandle: 400000000C07D9002000000000055962364961800B4FF9703...
  ⊞ obj_attributes  Regular File mode:0644 uid:0 gid:0
  ⊞ dir_attributes  Directory mode:0777 uid:0 gid:0
```

图10.21　针对 Lookup 请求的回应

11】 Frame12(如图 10.22 所示)：NFS 客户端发起一个 SetAttr Call 的请求，这个请求的目的是为了改变文件属性。可以看到客户端请求将文件的 atime 和 mtime 改为服务端当前的系统时间。

```
No.  Time        Source          Destination     Protocol  Info
  12 0.001129    10.128.132.45   10.128.132.175  NFS       V3 SETATTR Call, FH:0xf03ce91c
  13 0.001157    10.128.132.175  10.128.132.45   NFS       V3 SETATTR Reply (Call In 12)
⊞ Frame 12 (234 bytes on wire, 234 bytes captured)
⊞ Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
⊞ Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
⊞ Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 757, Ack: 873, Len: 168
⊞ Remote Procedure Call, Type:Call XID:0xe5f9fb18
⊟ Network File System, SETATTR Call FH:0xf03ce91c
    [Program Version: 3]
    [V3 Procedure: SETATTR (2)]
  ⊟ object
      length: 32
      [hash: 0xf03ce91c]
      decode type as: unknown
      filehandle: 400000000C07D9002000000000055962364961800B4FF9703...
  ⊟ new_attributes
    ⊞ mode: no value
    ⊞ uid: no value
    ⊞ gid: no value
    ⊞ size: no value
    ⊞ atime: set to server time
    ⊞ mtime: set to server time
  ⊞ guard: no value
```

图10.22　客户端的 SetAttr 请求

12】 Frame13(如图 10.23 所示)：对 Frame12 的回应。可以看到 atime、ctime 在之前和之后的不同，当然，时间差别都在微妙级，因为这一连串的请求其实是在很短的时间里发出去并得到应答的。

```
No.  Time        Source          Destination     Protocol  Info
  13 0.001157    10.128.132.175  10.128.132.45   NFS       V3 SETATTR Reply (Call In 12)
⊟ Network File System, SETATTR Reply
    [Program Version: 3]
    [V3 Procedure: SETATTR (2)]
    Status: NFS3_OK (0)
  ⊟ obj_wcc
    ⊟ before
        attributes_follow: value follows (1)
      ⊟ attributes
          size: 0
        ⊞ mtime: Aug 13, 2008 17:45:55.545410000
        ⊞ ctime: Aug 13, 2008 17:45:55.545410000
    ⊟ after  Regular File mode:0644 uid:0 gid:0
        attributes_follow: value follows (1)
      ⊟ attributes  Regular File mode:0644 uid:0 gid:0
          Type: Regular File (1)
        ⊞ mode: 0644
          nlink: 1
          uid: 0
          gid: 0
          size: 0
          used: 0
        ⊞ rdev: 0,0
          fsid: 0x000000000397ffb4
          fileid: 5608995
        ⊞ atime: Aug 13, 2008 17:45:55.545414000
        ⊞ mtime: Aug 13, 2008 17:45:55.545414000
        ⊞ ctime: Aug 13, 2008 17:45:55.545414000
```

图10.23　针对 SetAttr 请求的回应

13】 Frame15(如图 10.24 所示)：由于在客户端执行了“rm a”的命令，所以客户端发起一个对/mnt 目录的访问请求。其实客户端在抓包期间一直处于/mnt 目录中，至于为何要重新发起 Access 请求，与具体应用程序的代码有关，可能开发者为了确认/mnt 目录的 DH 值没有过期，所以重新试探访问。

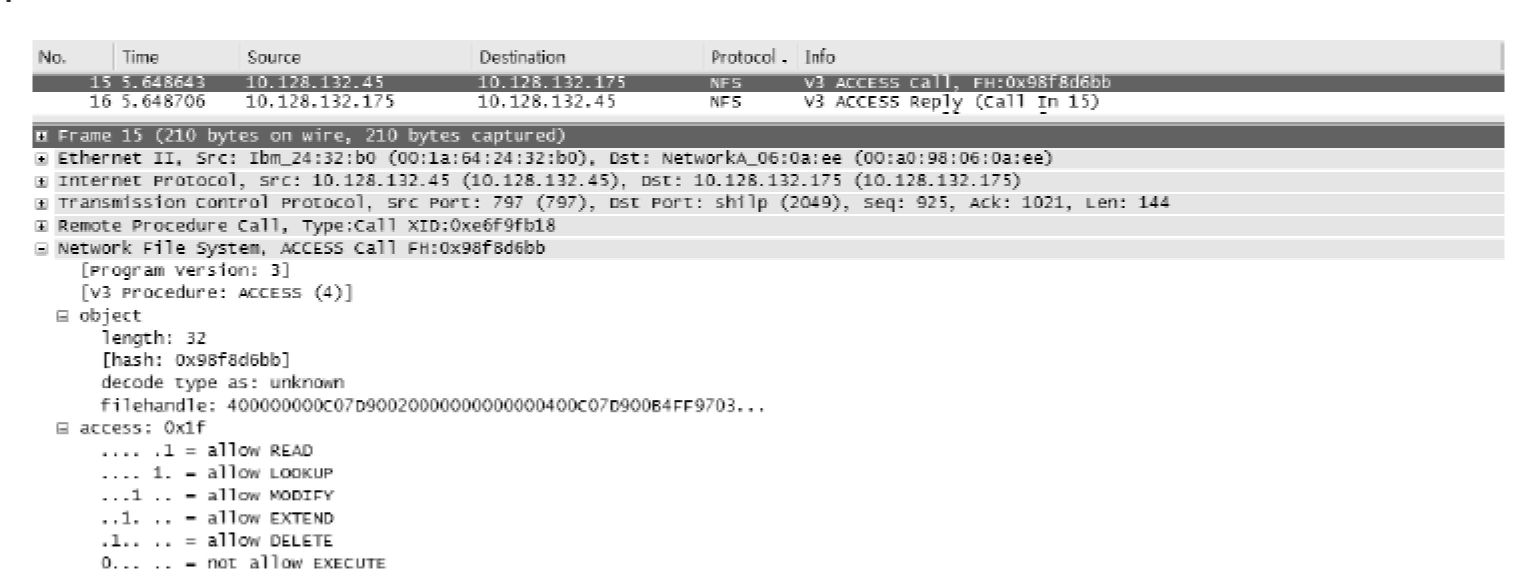


图10.24 客户端针对/mnt 目录的 Access 请求

14】 Frame16(如图 10.25 所示)：NFS 服务端对 Frame15 的回应。

```
No.  Time      Source          Destination     Protocol  Info
  15 5.648643  10.128.132.45   10.128.132.175  NFS       V3 ACCESS Call, FH:0x98f8d6bb
  16 5.648706  10.128.132.175  10.128.132.45   NFS       V3 ACCESS Reply (Call In 15)
Frame 16 (190 bytes on wire, 190 bytes captured)
Ethernet II, Src: NetworkA_06:0a:ee (00:a0:98:06:0a:ee), Dst: Ibm_24:32:b0 (00:1a:64:24:32:b0)
Internet Protocol, Src: 10.128.132.175 (10.128.132.175), Dst: 10.128.132.45 (10.128.132.45)
Transmission Control Protocol, Src Port: shilp (2049), Dst Port: 797 (797), Seq: 1021, Ack: 1069, Len: 124
Remote Procedure Call, Type:Reply XID:0xe6f9fb18
Network File System, ACCESS Reply
  [Program Version: 3]
  [V3 Procedure: ACCESS (4)]
  Status: NFS3_OK (0)
  obj_attributes  Directory mode:0777 uid:0 gid:0
    attributes_follow: value follows (1)
    attributes  Directory mode:0777 uid:0 gid:0
  access: 0x1f
    .... .1 = allow READ
    .... 1. = allow LOOKUP
    ...1 .. = allow MODIFY
    ..1. .. = allow EXTEND
    .1.. .. = allow DELETE
    0... .. = not allow EXECUTE
```

图10.25 针对 Access 请求的回应

15】 Frame18(如图 10.26 所示)：此时，客户端由于输入了“rm a”命令，所以客户端首先要查询一下文件“a”的权限，所以客户端发起一个 GetAttr 的请求来查询文件“a”的权限。

```
No.  Time      Source          Destination     Protocol  Info
  18 5.648902  10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0xf03ce91c
  19 5.648915  10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 18)  Regular File mode:0644 uid:0 gid:0
Frame 18 (206 bytes on wire, 206 bytes captured)
Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 1069, Ack: 1145, Len: 140
Remote Procedure Call, Type:Call XID:0xe7f9fb18
Network File System, GETATTR Call FH:0xf03ce91c
  [Program Version: 3]
  [V3 Procedure: GETATTR (1)]
  object
    length: 32
    [hash: 0xf03ce91c]
    decode type as: unknown
    filehandle: 400000000C07D900200000000055962364961800B4FF9703...
```

图10.26 客户端 GetAttr 请求

16】 Frame19(如图 10.27 所示)：NFS 服务端对 Frame18 的回应。

17】 Frame20(如图 10.28 所示)：NFS 客户端发起对文件“a”的 Access 请求。至于为何在删除文件之前要发起 Access 请求，与编码习惯有关。

18】 Frame21(如图 10.29 所示)：NFS 服务端对 Frame20 的回应。

19】 Frame23(如图 10.30 所示)：NFS 客户端发起 Remove 请求。可以看到 Remove 请求中并没有使用 FH，而是直接使用了文件名。

20】 Frame24(如图 10.31 所示)：NFS 服务端对 Frame23 的回应。

```
No.  Time      Source          Destination     Protocol  Info
  18 5.648902  10.128.132.45   10.128.132.175  NFS       V3 GETATTR Call, FH:0xf03ce91c
  19 5.648915  10.128.132.175  10.128.132.45   NFS       V3 GETATTR Reply (Call In 18)  Regular File mode:0644 uid:0 gid:0

⊞ Frame 19 (182 bytes on wire, 182 bytes captured)
⊞ Ethernet II, Src: NetworkA_06:0a:ee (00:a0:98:06:0a:ee), Dst: Ibm_24:32:b0 (00:1a:64:24:32:b0)
⊞ Internet Protocol, Src: 10.128.132.175 (10.128.132.175), Dst: 10.128.132.45 (10.128.132.45)
⊞ Transmission Control Protocol, Src Port: shilp (2049), Dst Port: 797 (797), Seq: 1145, Ack: 1209, Len: 116
⊞ Remote Procedure Call, Type:Reply XID:0xe7f9fb18
⊟ Network File System, GETATTR Reply  Regular File mode:0644 uid:0 gid:0
    [Program Version: 3]
    [V3 Procedure: GETATTR (1)]
    Status: NFS3_OK (0)
  ⊟ obj_attributes  Regular File mode:0644 uid:0 gid:0
      Type: Regular File (1)
    ⊞ mode: 0644
      nlink: 1
      uid: 0
      gid: 0
      size: 0
      used: 0
    ⊞ rdev: 0,0
      fsid: 0x000000000397ffb4
      fileid: 5608995
    ⊞ atime: Aug 13, 2008 17:45:55.545414000
    ⊞ mtime: Aug 13, 2008 17:45:55.545414000
    ⊞ ctime: Aug 13, 2008 17:45:55.545414000
```

图 10.27　针对 GetAttr 请求的回应

```
No.  Time      Source          Destination     Protocol  Info
  20 5.649109  10.128.132.45   10.128.132.175  NFS       V3 ACCESS Call, FH:0xf03ce91c
  21 5.649120  10.128.132.175  10.128.132.45   NFS       V3 ACCESS Reply (Call In 20)

⊞ Frame 20 (210 bytes on wire, 210 bytes captured)
⊞ Ethernet II, Src: Ibm_24:32:b0 (00:1a:64:24:32:b0), Dst: NetworkA_06:0a:ee (00:a0:98:06:0a:ee)
⊞ Internet Protocol, Src: 10.128.132.45 (10.128.132.45), Dst: 10.128.132.175 (10.128.132.175)
⊞ Transmission Control Protocol, Src Port: 797 (797), Dst Port: shilp (2049), Seq: 1209, Ack: 1261, Len: 144
⊞ Remote Procedure Call, Type:Call XID:0xe8f9fb18
⊟ Network File System, ACCESS Call FH:0xf03ce91c
    [Program Version: 3]
    [V3 Procedure: ACCESS (4)]
  ⊟ object
      length: 32
      [hash: 0xf03ce91c]
      decode type as: unknown
      filehandle: 400000000C07D900200000000055962364961800084FF9703...
  ⊞ access: 0x2d
```

图 10.28　客户端的 Access 请求

```
No.  Time      Source          Destination     Protocol  Info
  20 5.649109  10.128.132.45   10.128.132.175  NFS       V3 ACCESS Call, FH:0xf03ce91c
  21 5.649120  10.128.132.175  10.128.132.45   NFS       V3 ACCESS Reply (Call In 20)

⊞ Frame 21 (190 bytes on wire, 190 bytes captured)
⊞ Ethernet II, Src: NetworkA_06:0a:ee (00:a0:98:06:0a:ee), Dst: Ibm_24:32:b0 (00:1a:64:24:32:b0)
⊞ Internet Protocol, Src: 10.128.132.175 (10.128.132.175), Dst: 10.128.132.45 (10.128.132.45)
⊞ Transmission Control Protocol, Src Port: shilp (2049), Dst Port: 797 (797), Seq: 1261, Ack: 1353, Len: 124
⊞ Remote Procedure Call, Type:Reply XID:0xe8f9fb18
⊟ Network File System, ACCESS Reply
    [Program Version: 3]
    [V3 Procedure: ACCESS (4)]
    Status: NFS3_OK (0)
  ⊟ obj_attributes  Regular File mode:0644 uid:0 gid:0
      attributes_follow: value follows (1)
    ⊟ attributes  Regular File mode:0644 uid:0 gid:0
        Type: Regular File (1)
      ⊞ mode: 0644
        nlink: 1
        uid: 0
        gid: 0
        size: 0
        used: 0
      ⊞ rdev: 0,0
        fsid: 0x000000000397ffb4
        fileid: 5608995
      ⊞ atime: Aug 13, 2008 17:45:55.545414000
      ⊞ mtime: Aug 13, 2008 17:45:55.545414000
      ⊞ ctime: Aug 13, 2008 17:45:55.545414000
  ⊞ access: 0x2d
```

图 10.29　针对 Access 请求的回应

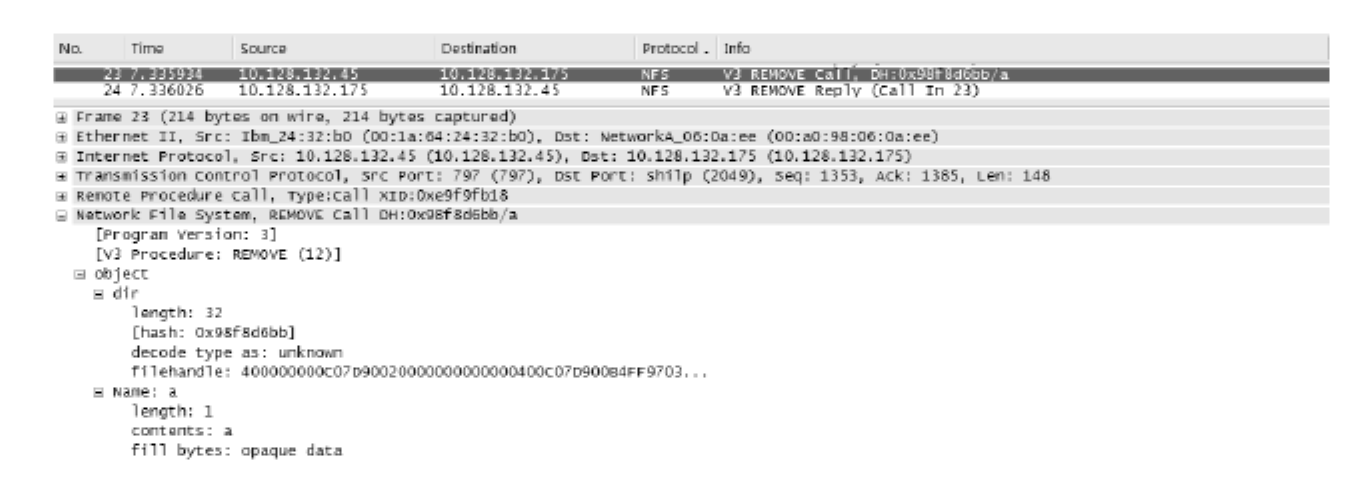


图 10.30　客户端的 Remove 请求

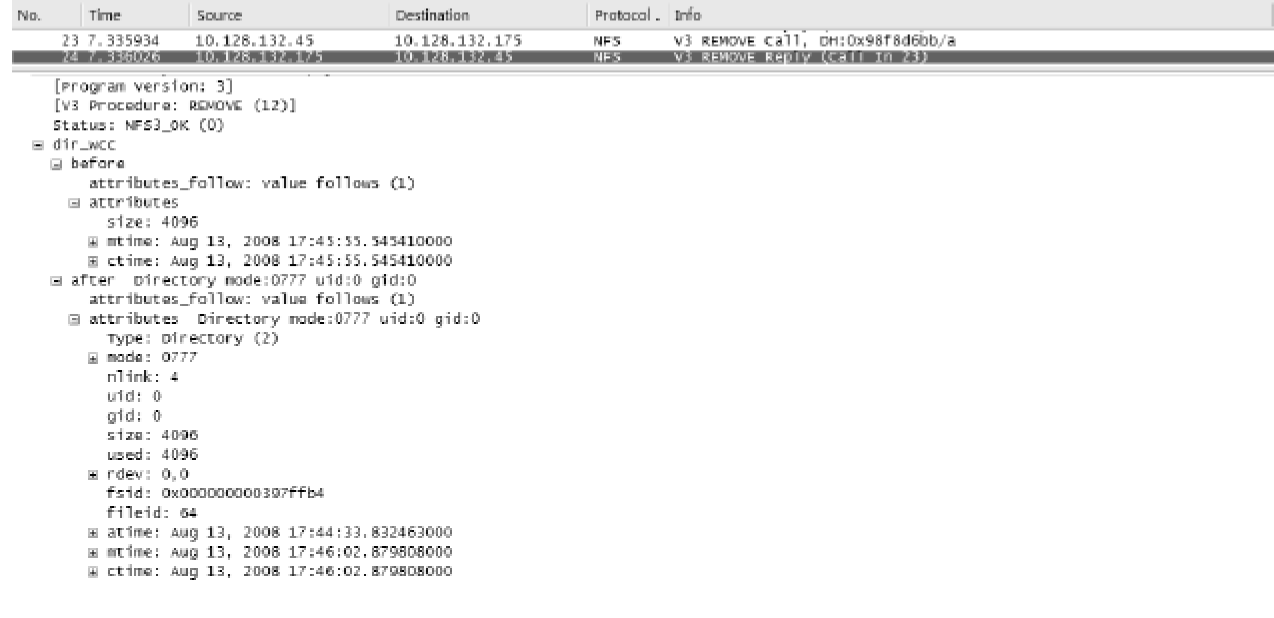


图 10.31　针对 Remove 请求的回应

可以看到，基于 NAS 的数据访问，客户端并不关心文件存放在磁盘的哪些扇区，这些逻辑全部由 NAS 服务端处理，客户端向 NAS 设备发送的只有各种文件操作请求以及实际的文件流式数据。大家可以在本书第 12 章第 4 节看到有关 ISCSI 的抓包分析，可以做一下对比，两者交互的语言完全不同。

10.1.4 美其名曰——NAS(Network Attached Storage 网络附加存储)

人们把这种带有集中式文件系统功能的盘阵，叫做网络附加存储(Network Attached Storage，NAS)。

> 提示
> NAS 不一定是盘阵，一台普通的主机就可以做成 NAS，只要它自己有磁盘和文件系统，而且对外提供访问其文件系统的接口(如 NFS、CIFS 等)，它就是一台 NAS。常用的 Windows 文件共享服务器就是利用 CIFS 作为调用接口协议的 NAS 设备。一般来说 NAS 其实就是处于以太网上的一台利用 NFS、CIFS 等网络文件系统的文件共享服务器。至于将来会不会有 FC 网络上的文件提供者，也就是 FC 网络上的 NAS，就要看是否有人尝试了。

1. SAN 和 NAS 的区别

前面说过，SAN 是一个网络上的磁盘，NAS 是一个网络上的文件系统。

> 提示
> 但是根据 SAN 的定义，即“存储区域网络”，SAN 其实只是一个网络，但是这个网络内包含着各种各样的元素，主机、适配器、网络交换机、磁盘阵列前端、盘阵后端、磁盘等。应该说，SAN 是一个最大的涵盖，它涵盖了一切后端存储相关的内容。所以从这个角度来看，SAN 包含了 NAS，因为 NAS 的意思是“网络附加存储”，它说的是一种网络存储方式，这么它就没有理由不属于 SAN 的范畴。所以，我认为 SAN 包含 NAS。
> 长时间以来，人们都用 SAN 来特指 FC，特指远端的磁盘。那么，好的，假设我设计出一种基于 FC 网络的 NAS，此时 SAN 代表什么呢？会发生滑稽的矛盾。但是，似乎还真想不出一种更简便更直观的叫法来称呼“FC 网络上的磁盘”这个事物。到此我也陷入定义的漩涡了，所以我们最好还是入乡随俗，本书之后的文字中，就把 FC 网络上的磁盘叫做 SAN，把以太网络上的文件系统称为 NAS。这里就是提一下，不要被表象所迷惑。

2. FTP 服务器为什么不属于 NAS

我们必须明白什么是网络文件系统，网络文件系统与本地文件系统的惟一区别，就是传输方式从主板上的导线变成了以太网络，其他方面包括调用的方式对于上层应用来说没有任何改变。

这就意味着，一旦用户挂载了一个网络文件系统目录到本地，那么他就可以像使用本地文件系统一样使用网络文件系统。

在 Windows 系统中，可以直接双击共享目录中的程序将其在本机运行(实际上是先通

过以太网将这个程序文件传输到本地的缓存，然后才在本地执行，而不是在远端执行)。而 FTP 无法做到这一点，FTP 不能实现诸如挂载等动作，它不是实时的。只有通过 FTP 将文件传输到本地的某个目录之后才能执行，而且这个程序执行需要的所有文件都必须在本地。

而网络文件系统则不然，即便某个本地执行的程序需要访问远端的某些文件，它也可以直接访问远端的文件，不需要预先将数据复制到本地再访问。所以，FTP、HTTP 和 TFTP 等文件服务并不属于网络文件系统，也不属于 NAS。

3. 普通台式机可以充当 NAS 么

完全可以，只要具备 NAS 的特性，就可以充当 NAS。

NAS 必须具备的物理条件如下。

- 不管用什么方式，NAS 必须可以访问卷或者物理磁盘。
- NAS 必须具有接入以太网的能力，也就是必须具备以太网卡。

普通台式机具备了这两个条件，就可以充当 NAS。我们只要编写程序从磁盘提取或者存放数据，记录好这些数据的组织方法，然后通过网络文件系统协议规定的格式进行发送或接收，就可以实现 NAS 的功能。或者可以直接在操作系统上编程，直接利用操作系统已经实现好的文件系统和网络适配器驱动程序，所要做的只是利用操作系统提供的足够简单的 API 编写网络文件系统的高层协议逻辑即可。

10.2 龙争虎斗——NAS 与 SAN 之争

1. SAN 快还是 NAS 快

很多人都在问，到底 SAN 快还是 NAS 快？要解答这个问题，方法非常简单，和百米赛跑一样，只要计算起点到终点的距离、耗时、开销就可以了。

SAN 的路径图如图 10.32 所示。

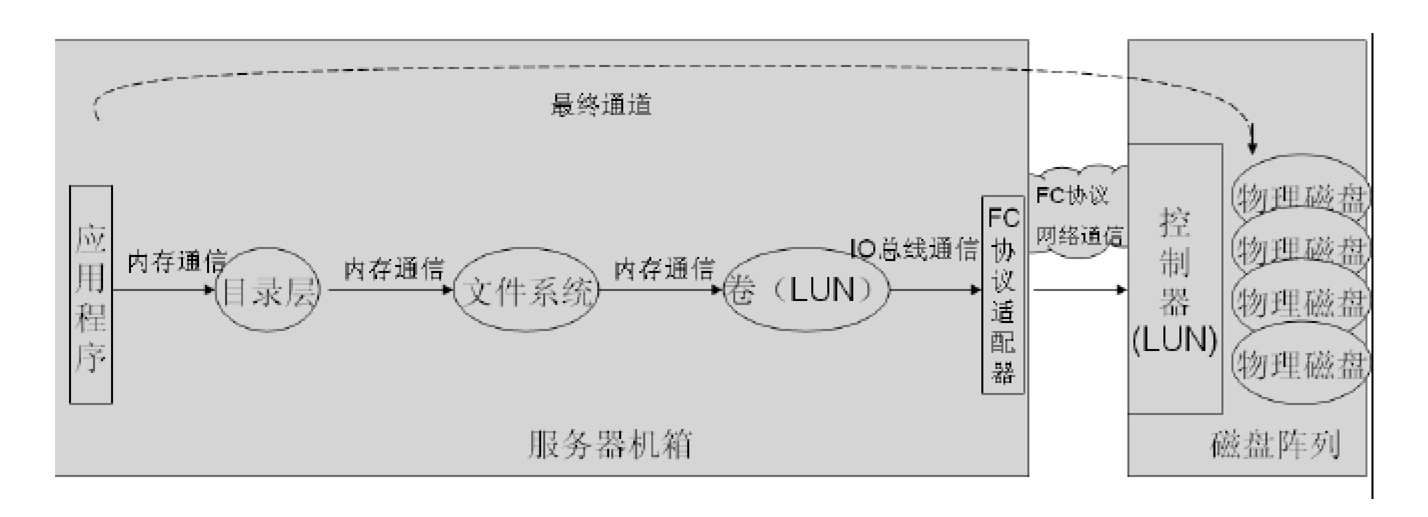


图10.32 SAN 方式路径图

NAS 的路径图如图 10.33 所示。

显然，NAS 架构的路径在虚拟目录层和文件系统层通信的时候，用以太网络和 TCP/IP 协议代替了内存，这样做不但增加了大量的 CPU 指令周期(TCP/IP 逻辑和以太网卡驱动程序)，而且使用了低速传输介质(内存速度要比以太网快得多)。而 SAN 方式下，路径中比 NAS 方式多了一次 FC 访问过程，但是 FC 的逻辑大部分都由适配卡上的硬件完成，增加不了多少 CPU 开销，而且 FC 访问的速度比以太网高。所以我们很容易的得出结论。如果后端磁盘没有瓶颈，那么除非 NAS 使用快于内存的网络方式与主机通信，否则其速度永远无法超越

SAN 架构。但是如果后端磁盘有瓶颈，那么 NAS 用网络代替内存的方法产生的性能降低就可以忽略。比如，在大量随机小块 IO、缓存命中率极低的环境下，后端磁盘系统寻道瓶颈达到最大，此时前端的 IO 指令都会处于等待状态，所以就算路径首端速度再快，也无济于事。此时，NAS 系统不但不比 SAN 慢，而且由于其优化的并发 IO 设计和基于文件访问而不是簇块访问的特性，反而可能比 SAN 性能高。

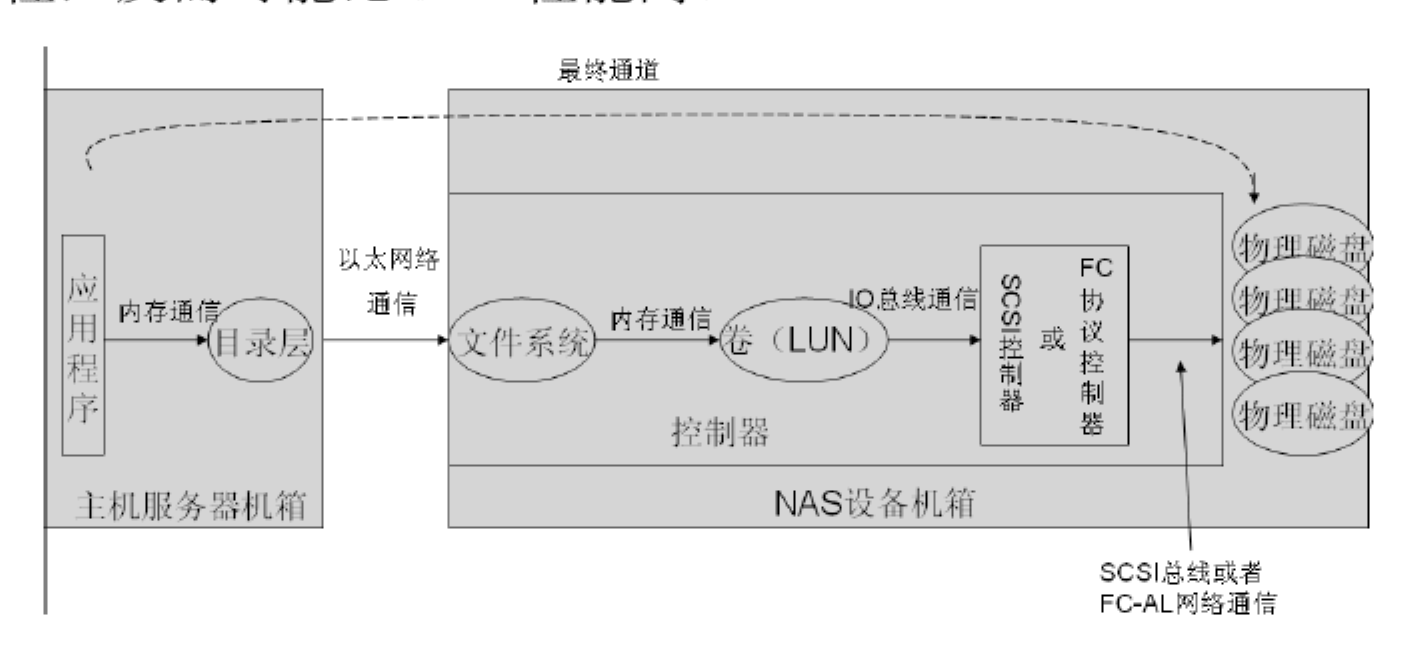


图10.33　NAS 方式路径图

既然 NAS 一般情况下不比 SAN 快，为何要让 NAS 诞生呢？既然 NAS 不如 SAN 速度快，那么它为何要存在呢？具体原因如下。

- NAS 的成本比 SAN 低很多。前端只使用以太网接口即可，FC 适配卡以及交换机的成本相对以太网卡和以太交换机来说是非常高的。
- NAS 可以解放主机服务器上的 CPU 和内存资源。因为文件系统的逻辑是要靠 CPU 的运算来完成的，同时文件系统还需要占用大量主机内存用作缓存。所以，NAS 适合用于 CPU 密集的应用环境。
- 由于基于以太网的 TCP/IP 传输数据，所以 NAS 可扩展性很强。只要有 IP 的地方，NAS 就可以提供服务，且容易部署和配置。
- NAS 设备一般都可以提供多种协议访问数据。网络文件系统只是其提供的一种接口而已，还有诸如 HTTP、FTP 等协议方式。而 SAN 只能使用 SCSI 协议访问。
- NAS 可以在一台盘阵上实现多台客户端的共享访问，包括同时访问某个目录或文件。而 SAN 方式下，除非所有的客户端都安装了专门的集群管理系统或集群文件系统模块，否则不能将某个 LUN 共享，强制共享将会损毁数据。
- 经过特别优化的 NAS 系统，可以同时并发处理大量客户端的请求，提供比 SAN 方式更方便的访问方法。
- 多台主机可以同时挂接 NFS 上的目录，那么相当于减少了整个系统中文件系统的处理流程，由原来的多个并行处理转化成了 NFS 上的单一实例，简化了系统冗余度。

2. SAN 好还是 NAS 好

关于 IO 密集和 CPU 密集说明如下。

- CPU 密集：指的是某种应用极其耗费 CPU 资源，其程序内部逻辑复杂，而且对磁盘访问量不高。如超频爱好者常用的 CPU 测试工具就是这种应用，这种程序在运行的时候，根本不用读取磁盘上的数据，只是在程序载入的时候，读入一点点程序数据而已。进程运行之后便会使 CPU 的核心处于全速状态，这会造成其他进程

在同一时间内只获得很少的执行时间，影响了其他程序的性能。在必要的时候，可以将多台机器组成集群来运行这种程序。

- IO 密集：指的是某种应用程序的内部逻辑并不复杂，耗费的 CPU 资源不多，但是要随时存取硬盘上的数据。比如 FTP 服务器之类程序就是这种。
- IO 和 CPU 同时密集：这种应用程序简直就是梦魇。为了获得高性能，大部分这类程序都不适合用单台机器运行，必须组成集群系统来运行这种应用程序，包括前端运算节点的集群和后端存储节点的集群。

显然，NAS 对于大块顺序 IO 密集的环境，要比 SAN 慢一大截，原因是积累效应。经过大量 IO 积累之后，总体差别就显现出来了。不过，如果要使用 10G 以太网这种高速网络，无疑要选用 NAS，因为底层链路的速度毕竟是目前 NAS 的根本瓶颈。此外，如果是高并发随机小块 IO 环境或者共享访问文件的环境，NAS 会表现出很强的相对性能。如果 SAN 主机上的文件系统碎片比较多，那么读写某个文件时便会产生随机小块 IO，而 NAS 自身文件系统会有很多优化设计，碎片相对少。CPU 密集的应用可以考虑使用 NAS。

SAN 与 NAS 各有各的优点和缺点，需要根据不同的环境和需求来综合考虑。

3. 与 SAN 设备的通信过程

对于 SAN 方式来说，应用程序必须通过运行在服务器本机或者 NAS 设备上的文件系统与磁盘阵列对话。应用程序对本机文件系统(或 NAS)说：“嗨，兄弟，帮我把/mnt/SAN 目录下的 SAN.txt 文件传到我的缓冲区。”文件系统开始计算 SAN.txt 文件占用的磁盘扇区的 LBA 地址，计算好之后向 SAN 磁盘阵列说(用 SCSI 语言)：“嗨，哥们，把从 LBA10000 开始之后的 128 个扇区内容全部传送给我！”

SAN 磁盘阵列接收到这个请求之后，便从它自身的众多磁盘中提取数据，通过 FC 网络传送给运行着文件系统程序的节点(服务器主机或 NAS 设备)。 文件系统接受到扇区内容之后，根据文件系统记录截掉扇区多余的部分，将整理后的数据放入请求这个数据的应用程序的缓冲区。

4. 与 NAS 设备的通信过程

应用程序通过操作系统的虚拟目录层直接与 NAS 设备对话：“嗨，兄弟，帮我把/mnt/NAS 目录下的 NAS.txt 文件传过来。”或 “嗨，兄弟，帮我把/mnt/NAS 目录下的 NAS.txt 文件的前 1024 字节传递过来。”这些话被封装成 TCP/IP 数据包，通过以太网传递到 NAS 设备上。NAS 接到这个请求之后，立即用自己的文件系统(NTFS、JFS2、EXT2、EXT3 等)计算 NAS.txt 文件都占用了磁盘的哪些扇区，然后从自己的磁盘上用 ATA 语言或者 SCSI 语言向对应的磁盘存取数据(或自己安装 FC 适配卡，从 SAN 存储设备上存取数据)。

显然，NAS 将文件系统逻辑搬出了主机服务器，成为了一个单独的文件系统逻辑运行者。

5. 文件提供者

NAS 可以看作是一个 Filer。Filer 这个词是著名 NAS 设备厂商 NetApp 对其 NAS 产品的通俗称呼。它专门处理文件系统逻辑及其下面各层的逻辑，从而解放了服务器主机。服务器主机上不必运行文件系统逻辑，甚至也不用运行磁盘卷逻辑，只需要运行目录层逻辑(UNIX 系统上 VFS 层、Windows 系统上的盘符及目录)即可。把底层的模块全部交由一个独

立的设备来完成，这样就节约了服务器主机的 CPU 资源和内存资源，从而可以专心的处理应用层逻辑了。

NAS 网关就是这样一种思想。NAS 网关其实就是一台运行文件系统逻辑和卷逻辑的设备，可以把它想象成一个泵，这个泵可以从后端接收一种格式(以 LBA 地址为语言的指令和数据格式)，经过处理后从前端用另一种格式(以文件系统为语言的指令和数据格式)发送出去，或执行反向的过程。可以把这个泵接入任何符合条件的网络中，以实现它的功能。我们可以称它为文件系统泵，或者 Filer。我们把 SAN 设备称为 Disker(专门处理磁盘卷逻辑)，把服务器主机称为 Applicationer(专门处理应用逻辑)。如果某个设备集成了 Filer 和 Disker 的功能，并将其放入了一个机箱或者机柜，那么这个设备就是一个独立的 NAS 设备。如果某个设备仅仅实现了 Filer，而 Disker 是另外的独立设备，这个只实现了 Filer 的设备就称为 NAS 网关或 NAS 泵。

图 10.34 显示了这个泵接入网络之后发生的变化。

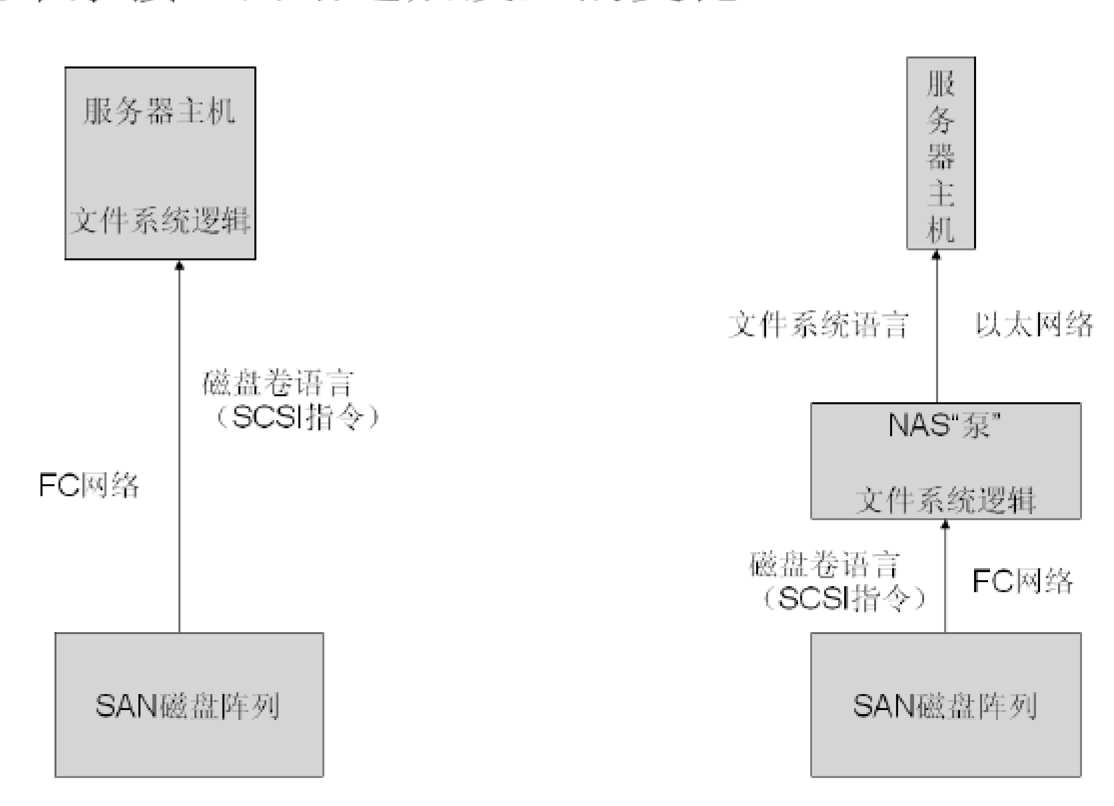


图10.34　NAS 泵

然而，目前 NAS 的用途并不如 SAN 广泛，主要原因是因为 NAS 的前端接口几乎都是千兆以太网接口，而千兆以太网的速度也不过 100MB/s，除去开销之后所剩无几。而 SAN 设备的前端接口目前普遍都是 4Gb/s 的速度，可以提供 400MB/s 的带宽。FC 现在已有 8Gb/s 速率的接口出现，而 10Gb/s 以太网也初露端倪。不久的将来，NAS 必定会发起另一轮进攻。

10.3　三足鼎立——DAS、SAN 和 NAS

人们将最原始的存储架构称为 DAS，即 Direct(Dedicate)Attached Storage(直接连接存储)，意思是指存储设备只用于与独立的一台主机服务器连接，其他主机不能使用这个存储设备。如 PC 里的磁盘或只有一个外部 SCSI 接口的 JBOD 都属于 DAS 架构。

纵观武当仓库的改革过程，恰恰正是一个从 DAS(仅供自己使用)，到 SAN(出租仓库给其他租户使用)，再到 NAS(集中式理货服务外包)的过程。

到此，DAS、SAN、NAS 形成了存储架构的三大阵营，且各有其适用条件，形成了三足鼎立之势！

但是，大家一定要牢记，SAN(Storage Area Network)是一种网络，而不是某种设备。只要是专门用来向服务器传输数据的网络都可以被称为 SAN。所以，NAS 设备使用以太网

络向主机提供文件级别的数据访问，那么以太网络就是 SAN。由于在高端领域 NAS 的使用不如 FC SAN 设备多，所以人们习惯的称 FC SAN 架构为 SAN。我们也顺从习惯，但是一定要明白其中的门道。

10.4 最终幻想——将文件系统语言承载于 FC 网络传输

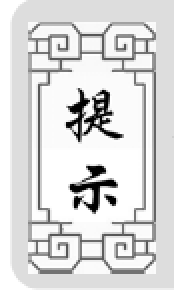
既然 SCSI 语言及数据可以用 FC 协议传递，文件系统语言可以用以太网传递，那么文件系统语言能不能用 FC 传递呢？再者 SCSI 语言能不能通过以太网来传递呢？完全可以，而且后者已经被实现并被广泛应用了，本书第 12 章讲的就是它。在四种组合中，就差将文件系统语言承载于 FC 网络传输这个想法没有被实现，或者说有人实现了但是没有听说。

现在来看前者。试想一下，如果用 FC 协议传递文件系统语言和数据而不是磁盘卷语言及其数据，那么开销会小很多。文件系统语言毕竟是比较上层和高级的语言，相对于底层磁盘卷语言来说，它非常简单。

读写某个文件，如果用高层语言只需要描述关于这个文件的信息即可。但是如果使用低级语言，可能要发送很多的语句。如果这个文件在磁盘上形成了很多碎片的话，发送的 IO 指令将不计其数。这就像 C 语言被编译器编译成汇编语句一样，文件系统语言是高级语言，比如“将 C 盘下 C.txt 文件传送过来”这一句高级语言，会被翻译成更低级的多条 SCSI 语言。利用比内存总线速度慢得多的 FC 网络来传送这些低级语言，无疑是非常浪费资源的。低级语言就应该在内存中传递而不是外部低速网络上，这样才能达到性能最大化。

目前普遍的架构是文件系统和磁盘控制器驱动程序都运行在应用服务器主机上。文件系统向卷发送的请求是通过内存来传递的，而主机向磁盘(LUN)发送的请求是通过 FC 网络来传递的，后者速度显然比前者慢很多。如果将“文件系统、磁盘控制器和磁盘(SAN 盘阵或者本地磁盘)”这三个部件，整体搬到应用服务器主机外部，成为一个独立的 NAS 系统。然后这个 NAS 设备用 FC 协议与服务器主机通信，FC 协议上承载的是文件系统语言，这就可以既保证降低了服务器的开销，又保证了数据传输速度。

也许受限于技术或者商业和市场的限制，File system(或 File) over FC(FC 网络文件系统)始终没有被提出或开发出来。不过在 10G 以太网普及之后，相信 FC 设备也会因为高成本高专用化、兼容性等问题逐渐被淘汰。目前市场上的 NAS 设备普遍采用以太网加上 TCP/IP 的模式来传送文件系统指令，我们可以称这种架构为“File system over Ethernet(FSoE)”。



关于协议之间相互杂交的论题，在本书后面会详细讨论。

10.5 长路漫漫——系统架构进化过程

下面总结一下系统架构进化过程中的 9 个阶段。

10.5.1 第一阶段：全整合阶段

图 10.35 是一个最原始的普通服务器的架构，所有部件和模块都在一个服务器机箱中。这属于 DAS 架构，因为主机机箱内的磁盘只被本机使用。

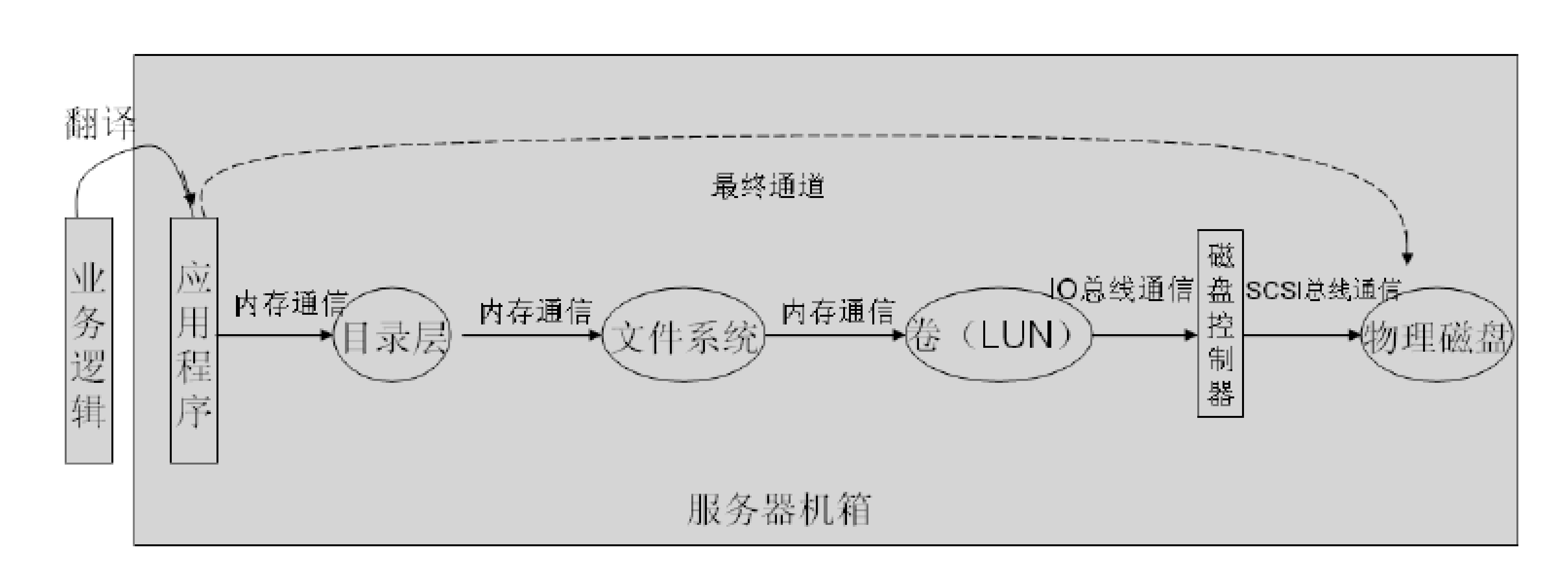


图10.35 全整合阶段

10.5.2 第二阶段：磁盘外置阶段

图 10.36 所示的架构，是将磁盘置于服务器机箱外部的情况。这种架构依然属于 DAS 架构，因为存储系统只被一台主机使用。

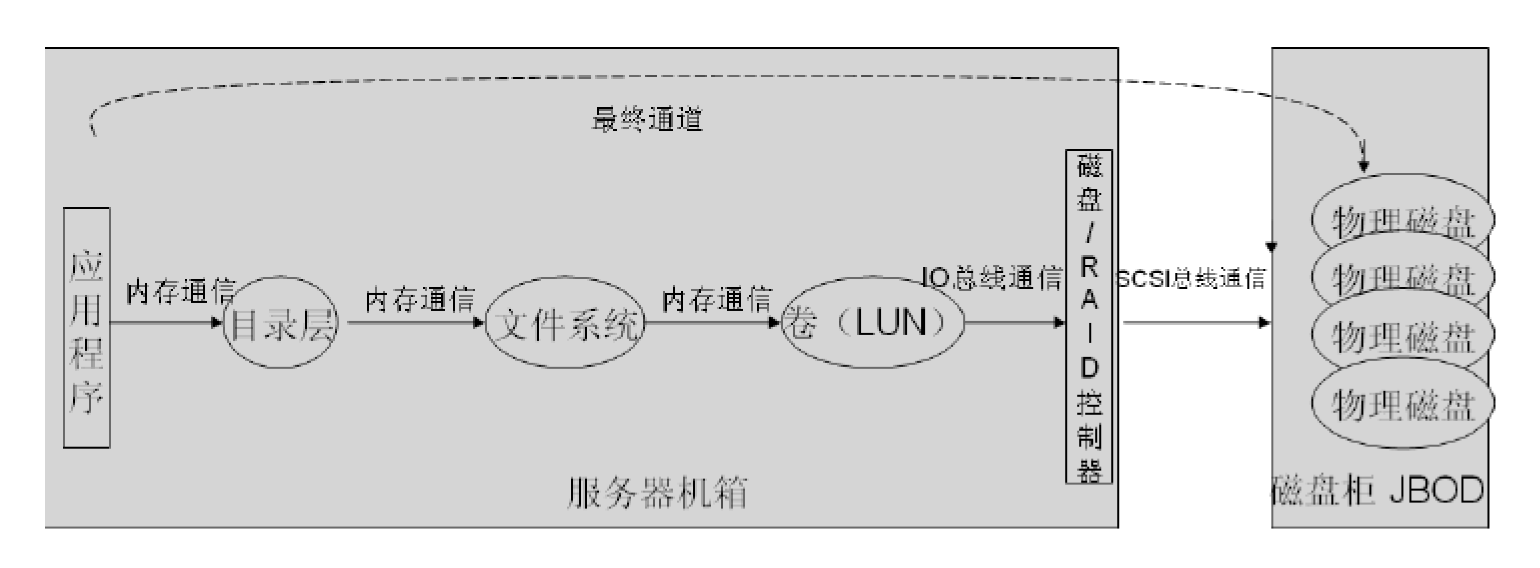


图10.36 磁盘外置阶段

10.5.3 第三阶段：外部独立磁盘阵列阶段

图 10.37 是服务器主机通过普通 SCSI 线缆连接外部独立磁盘阵列的情况。这种简单的 SCSI 接口盘阵只能供一台或者几台(如果盘阵提供多个外部 SCSI 接口的话)主机接入，称不上彻底的网络化，但可以被称为 SAN，因为这种架构已经开始显现网络化萌芽了。

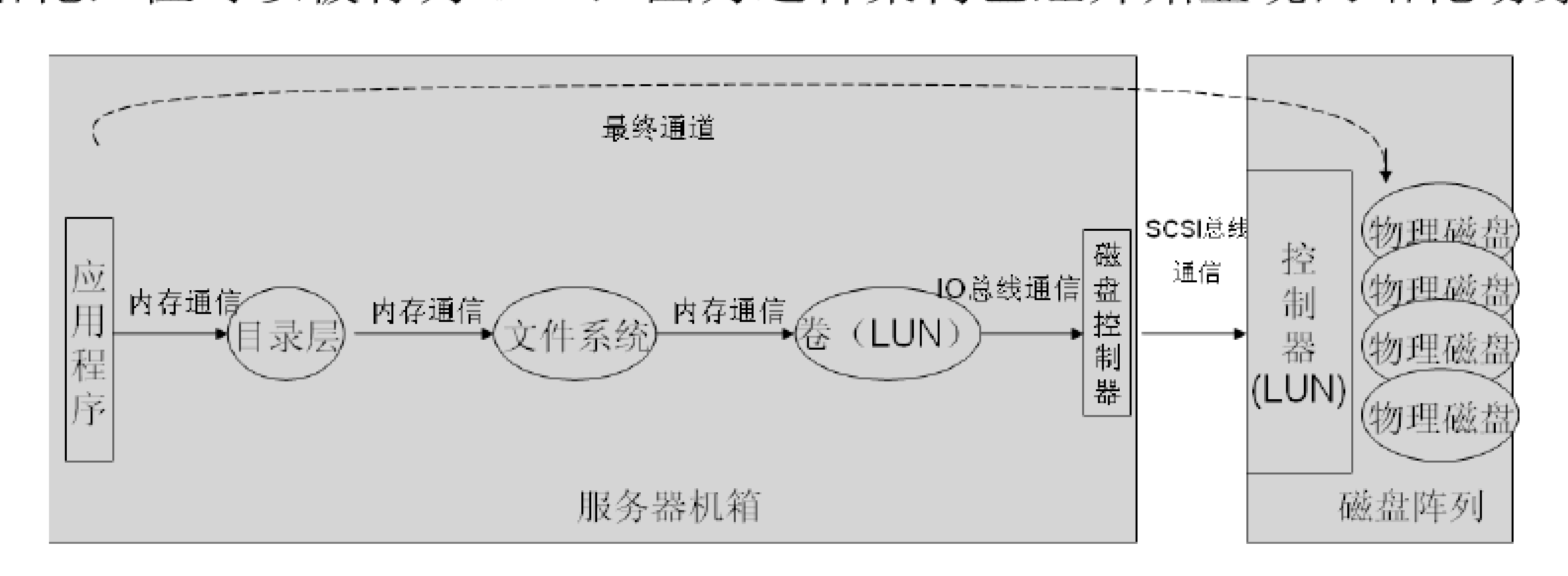


图10.37 外部独立磁盘阵列阶段

10.5.4 第四阶段：网络化独立磁盘阵列阶段

图 10.38 是一台服务器用 FC 网络连接 FC 接口磁盘阵列的情况。图中磁盘阵列真正成为包交换网络上的一个节点，可同时被多个其他节点访问，是向彻底网络化进化的里程碑。这种架构是彻彻底底的 SAN 架构。

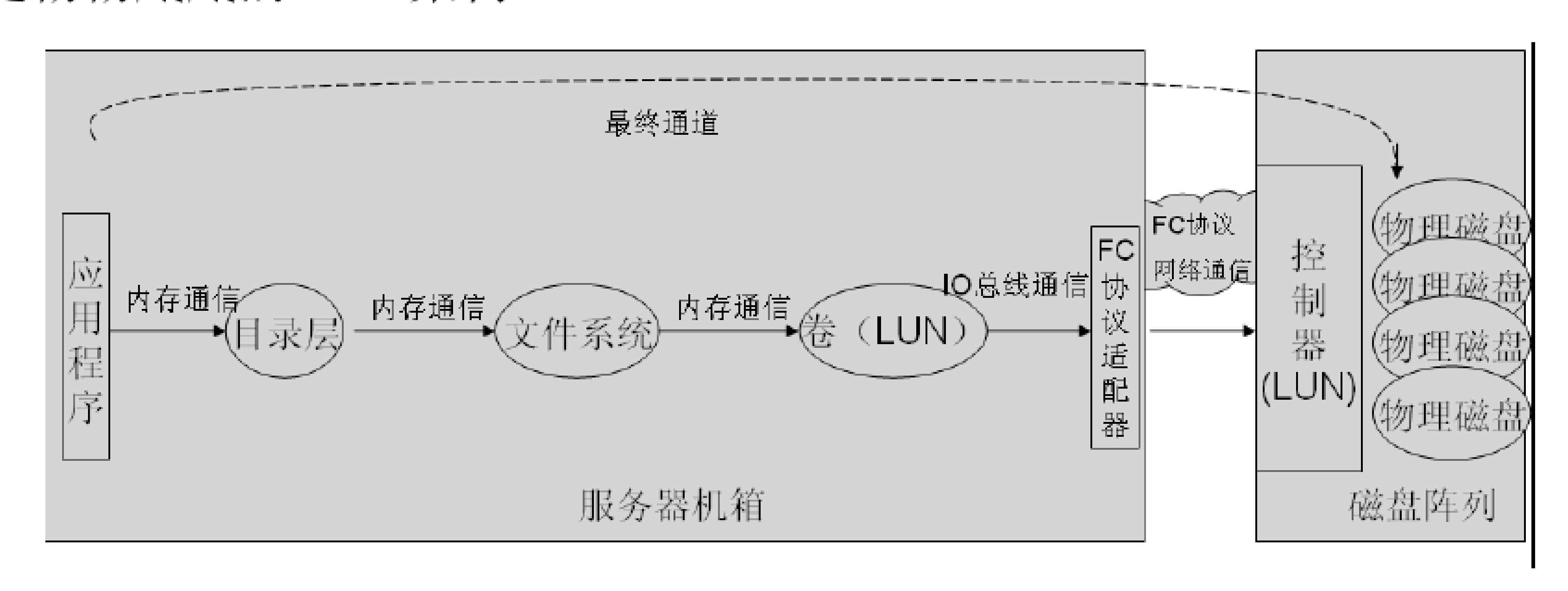


图10.38 网络化独立磁盘阵列阶段

10.5.5 第五阶段：瘦服务器主机、独立 NAS 阶段

图 10.39 中，服务器主机用以太网与 NAS 设备进行通信从而存储数据。在瘦服务器阶段，文件系统之下的所有层次模块位移到了外部独立设备中。主机得到了彻底的解放，专门处理业务逻辑，而不必花费太多资源去处理底层系统逻辑。

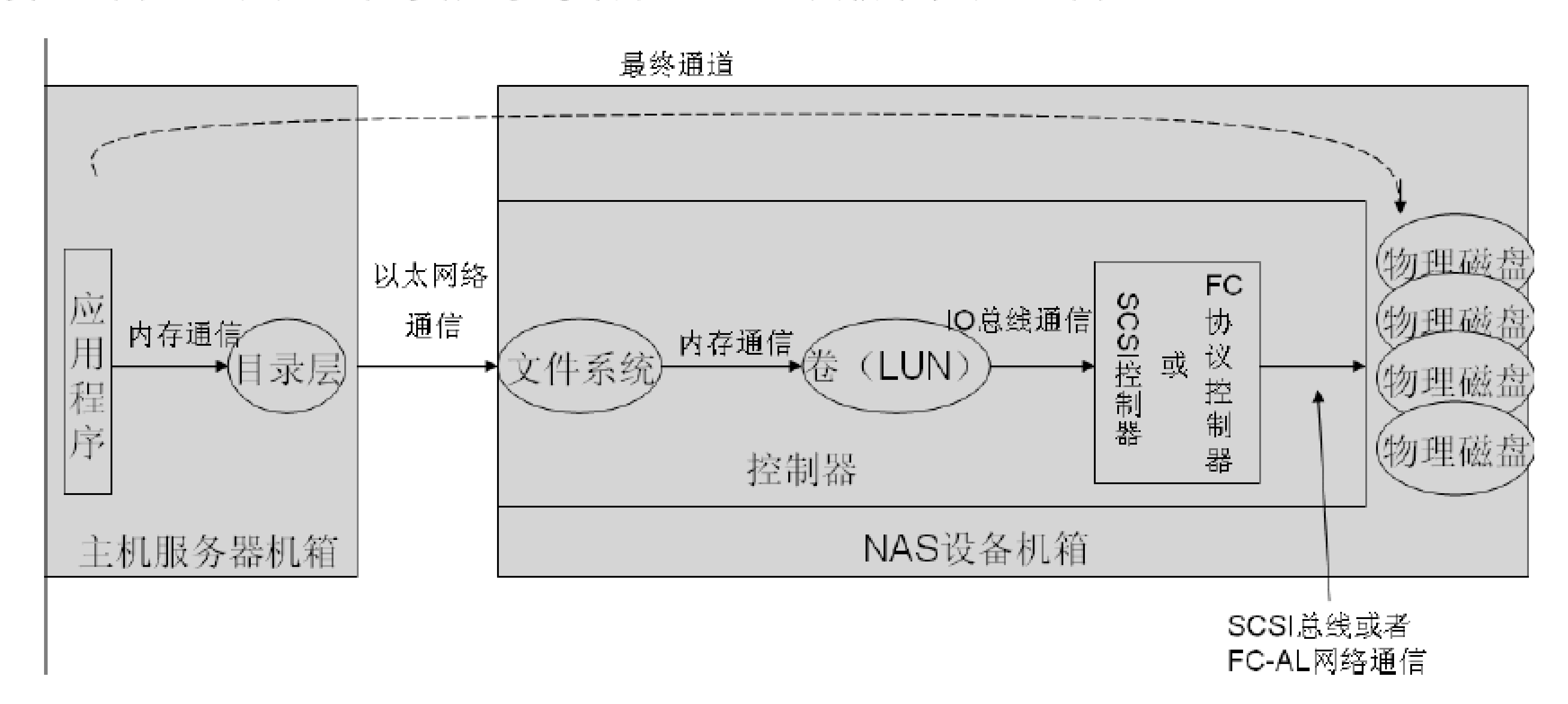


图10.39 瘦服务器独立 NAS 阶段

10.5.6 第六阶段：全分离式架构

在 NAS 设备的后端可以用包含在自己机箱内的硬盘，也可以用并行 SCSI 来连接磁盘，还可以用 FC 协议来连接 SAN 盘阵来获得 LUN(NAS 网关)。在这个阶段中，所有部件彻底地分离了，每个部件都各伺其职，中间通过不同的网络方式通信，前端用以太网，后端用 FC 网，如图 10.40 所示。

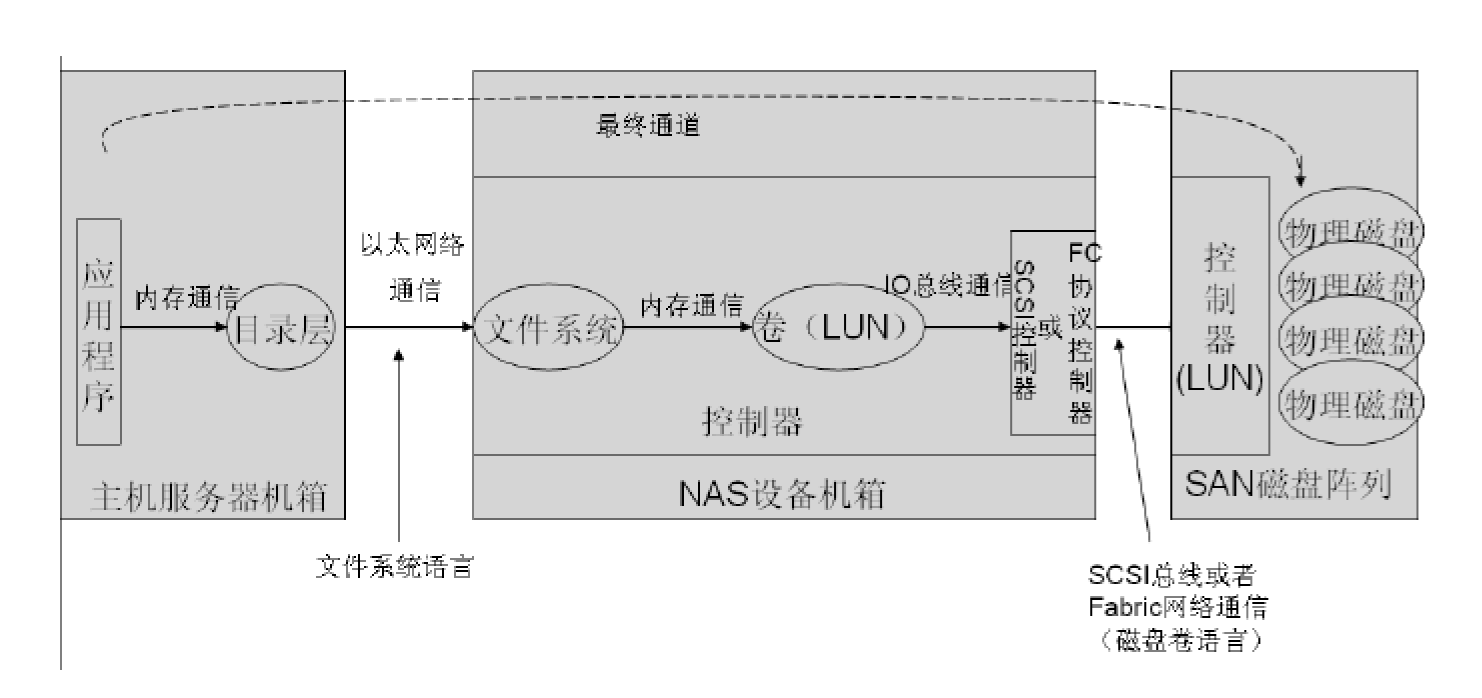


图10.40　全分离阶段

10.5.7　第七阶段：能量积聚，混沌阶段

在图 10.41 的环境中，既有纯 SAN 的磁盘阵列，来利用自己机箱或扩展柜内的磁盘，又有 NAS 网关设备，不但可以利用本地磁盘，还可以向 SAN 磁盘阵列“租赁”若干 LUN 卷。另外还有一台多协议磁盘阵列，它既可以向外提供 FC 协议的连接方式(承载磁盘卷语言)，又可以提供以太网的连接方式(承载文件系统语言)，这种设备是 SAN 与 NAS 融合的结果。服务器可以选择直接用磁盘卷语言访问 SAN 磁盘阵列上的 LUN，用运行在服务器上的文件系统程序管理磁盘卷，也可以选择直接通过以太网络访问 NAS 设备上的目录，用文件系统语言向 NAS 发送指令。

这个架构最终就是一个混沌的架构，既有纯 SAN 和纯 NAS，又有融合的 SAN 和 NAS。第八阶段也是目前 IT 系统所应用的架构。

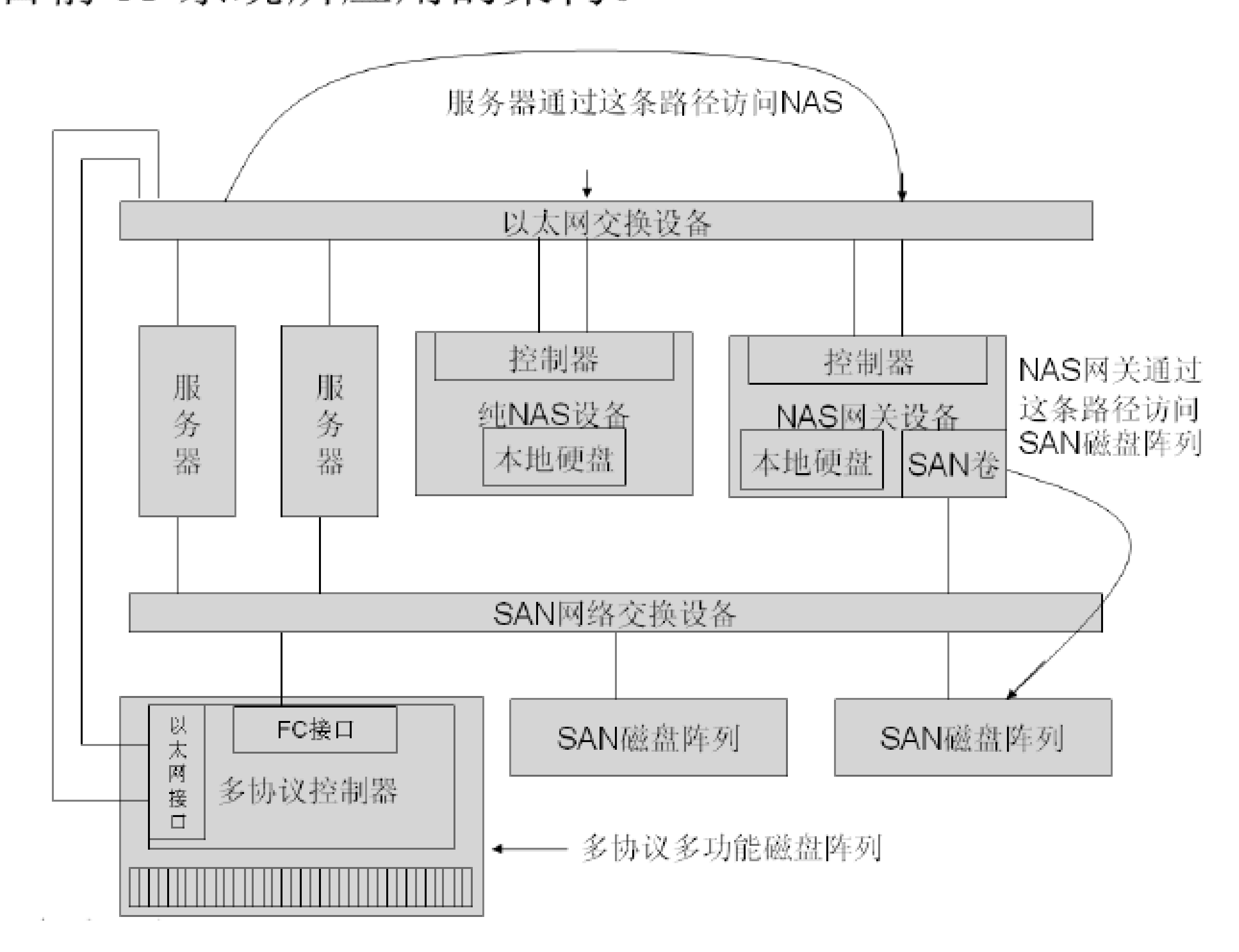


图10.41　高能混沌阶段

10.5.8　第八阶段：收缩阶段

当机械硬盘被芯片所取代，并且大规模集成电路技术不断地发展，操作系统中的所有底层模块均用专用硬件电路实现。CPU 所做的仅仅是进行业务逻辑计算。外部接口已经过渡到了全部使用无线网络进行通信，包括连接显示器(或许已经被 3D 眼镜所取代)。

目前，Intel 公司已经在其 CPU 中集成了显示芯片，使计算机不再需要加装独立的显卡，也不用占用主板上的空间来集成显示芯片。Sun 公司将多个万兆以太网芯片集成在了最新的 CPU 上。Bluearc 公司的存储产品更是已经将整个操作系统的所有逻辑都做到了 FPGA 芯片中。这些案例很好的证明了 IT 架构正在收缩之中。图 10.42 所示为收缩阶段系统架构示意图。

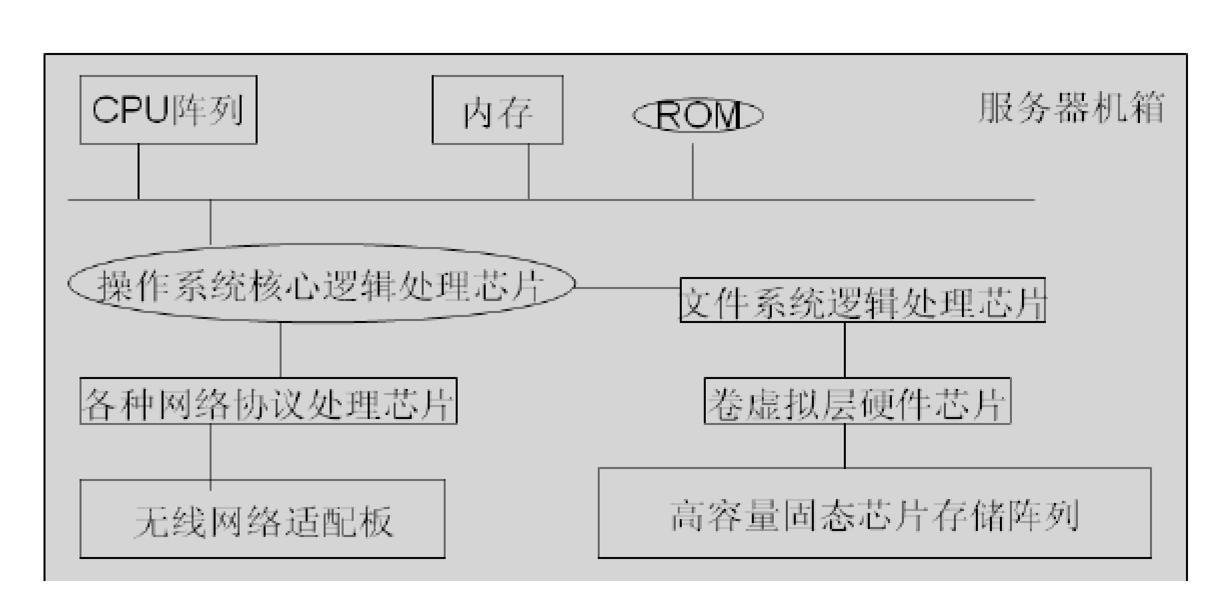


图10.42 收缩阶段

10.5.9 第九阶段：强烈坍缩阶段

对于系统架构，坍缩之后的 IT 系统会节约大量空间和电力能量。随着集成电路技术发展到一定程度之后，甚至可以将整个系统集成于一个微小的芯片中，即所谓的 System On Chip(SOC)。IT 系统将坍缩到这个芯片中，如图 10.43 所示。

纵观这 9 个阶段，从全整合阶段(占用很大空间和资源)，到坍缩阶段(极小的空间极高的质量)，这正是宇宙发展演化的规律。在一个极大的空间中包含了众多物质，然后通过不断演化，最终坍缩到一个小空间内，同样包含了极大的质量。



图10.43 强烈坍缩阶段

到此，SAN 已经不仅仅限于通过 FC 网络传递 SCSI 指令的架构了。SAN 的概念，既然是“存储区域网络”的意思，那么就应该泛指参与主机服务器后端存储系统的所有部件，甚至包括 NAS，我们可以将 NAS 看作 SAN 的一个分支架构。

SAN 在 IT 系统架构进化到第九阶段之前还会继续存在。在第九阶段，SAN 的概念或许会被赋予另外的意义。

10.6 泰山北斗——NetApp 的 NAS 产品

NetApp 公司掌握了全球最先进的 NAS 方面的相关技术，它的 FAS 系列产品统治了 NAS 市场的大部江山。FAS 系列中的所有产品均运行 Data ONTAP 操作系统，这是 NetApp 专门开发的针对 NAS 的操作系统。

既然是 NAS，其内部的文件系统层肯定是一个功能强大而稳定的层次。ONTAP 系统中的文件系统名为 WAFL，这是一个充满个性的文件系统。

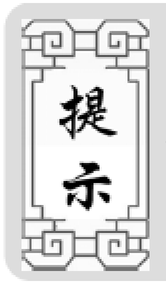
NetApp 自称其存储产品为“Filer”，下面就来看看 Filer 的四把杀手锏。

10.6.1 WAFL 配合 RAID 4

Write Anywhere Filesystem Layout(WAFL)是 NetApp 公司开发的一种文件系统。这个文件系统最大的特点，也是其他所有文件系统都没有实现的特点，就是它能按照 RAID 4 的喜好来向 RAID 4 卷写数据。RAID 4 由于其独立校验盘的设计，导致它只能接受顺序的写入 IO 而不能并发，所以它对于其他厂家的盘阵来说完全就是一个灾星，没有人敢用，也没有人愿意用。然而 NetApp 偏偏采用了它，而且通过 WAFL 的调教，RAID 4 在 FAS 产品上发挥出了很好的性能。WAFL 是怎么调教 RAID 4 的呢？

与其说是 WAFL 调教 RAID 4，不如说是 RAID 4 逼迫 WAFL 就范。

RAID 4 再不好也有可取的地方，如果 IO 写入有很大几率是整条写的形式，那么 RAID 4 便会表现得像 RAID 0 一样良好。不仅仅是 RAID 4，任何校验型的 RAID，如 RAID 5、RAID 3，只要是整条写，便会产生极高的性能。



关于整条写的概念，请参考本书第 4 章的内容。

既然校验型 RAID 最喜欢被整条写，那么就不妨满足这个要求。WAFL 就是这么被设计出来的。

图 10.44 所示的是一个 5 块盘组成的 RAID 4 系统。条带大小 20KB，条带在每个数据盘上分割出的 segment 大小为 4KB。假如某一时刻应用要求 WAFL 写入三个文件，如 /tmp/file1、/tmp/file2 和 /tmp/file3，其中 file1 大小为 4KB，file2 大小为 8KB，file3 大小为 4KB。则 WAFL 便会计算出：如果将这三个文件对应的数据写到一整条条带上，就构成了整条写，性能得到了提升。

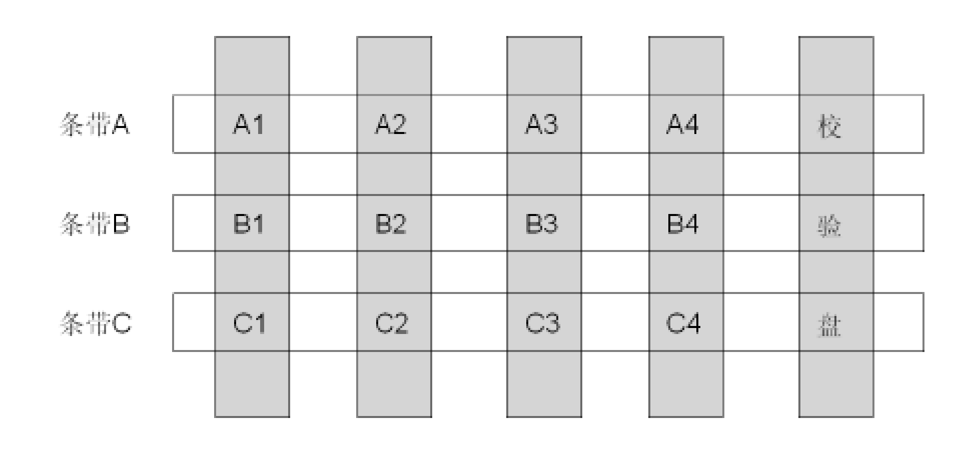


图 10.44　5 块盘组成的 RAID 系统

所以，WAFL 先在其元数据中做好记录：/tmp/file1 对应数据块为 A1，/tmp/file2 对应数据块为 A2+A3，/tmp/file3 对应数据块为 A4。然后将这三个文件的数据合并写入 RAID。当然，如果要求增加高并发度，那么 WAFL 也可以将同一个文件对应的一定长度的块写入同一个硬盘。

实际上，文件系统就应该适配底层的特性，只有这样才能获得最优的性能。普通文件系统中，在选取空闲块写入数据的时候，并没有针对底层的 RAID 级别来做对应的优化。而 WAFL 中有一个专门的 Write Allocation 模块来负责优化。WAFL 的做法无疑对文件系统的优化起到了领头和示范作用。

以上只是简略说明 WAFL 的思想，实际操作中还需要考虑诸如元数据的写入、块的偏移等很多复杂情况。

10.6.2 Data ONTAP 利用了数据库管理系统的设计

我们知道，数据库管理系统是这样记录日志的：在某时刻，数据库管理系统接收到应用程序的 SQL 更新语句及其对应的数据，在将这些数据更新到缓存中覆盖原有数据的同时，将这个操作的动作以及对应的数据，以日志的形式记录到位于内存的日志缓冲区内。每当应用程序发起提交指令或每隔几秒钟的时间，缓冲区内的日志就会被写入到磁盘上的日志文件中，以防止意外掉电后造成的数据不一致性，同时将缓存中更新过的数据块写入磁盘。只有当日志被确实的写入到硬盘上的日志文件中后，数据库管理程序才会对上层应用返回执行成功的信号。

数据系统是一个非常复杂的系统，也是一个对数据一致性要求极高的系统，所以数据库利用记录操作日志的方式来保证数据一致性的做法是目前普遍使用，而且是实际效果最好的方法。NetApp 的 Data ONTAP 操作系统，就是利用了数据库这种设计思想，它把向文件系统或卷的一切写入请求作为操作日志记录到 NVRAM 中保存，每当日志被保存到了 NVRAM 中，就会向上层应用返回写入成功信号。

为何要用 NVRAM 而不是文件来保存日志呢？

10.6.3 利用 NVRAM 来记录操作日志

数据库系统完全可以直接将操作日志写入磁盘，而不必先写入内存中的日志缓冲区，再在触发条件下将日志写入磁盘上的文件。然而这么做会严重降低性能，因为日志的写入是非常频繁的，且必须为同步写入。如果每条日志记录都写入磁盘，则由于磁盘相对于内存来说是慢速 IO 设备，所以会造成严重的 IO 瓶颈。所以必须使用内存中的一小块来作为日志缓存。

但是一旦系统发生意外掉电，则内存中的日志还没来得及保存到硬盘就会丢失。在数据库再次启动之后，会提取硬盘上已经保存的日志文件中的条目来重放这些操作，对于没有提交的操作，进行回滚。这样，就保证了数据的一致性。

如果某个应用程序频繁的进行提交操作，则日志缓冲区的日志便会被频繁写入磁盘。在这种情况下，日志缓存就起不到多少作用了。幸好，对于很多需要访问数据库的应用程序来说，上层的每个业务操作一般都算作一个交易，在交易尚未完成之前，程序是不会发送提交指令给数据库系统的，所以频繁提交发生的频率不高。

然而，上层应用向文件系统中写数据的话，每个请求都是一次完整的交易。如果对每个请求都将其对应的操作日志写入磁盘，开销就会很大。所以 NetApp 索性利用了带电池保护的 RAM 内存用作记录操作日志的存储空间。这样一来，不但日志写入的速度很快，而且不用担心意外掉电。有了这种电池保护的 RAM(称为 NVRAM 似乎不合适，因为 NVRAM 不用电池就可以在不供电的情况下保存数据，而 NetApp 使用的是带电池保护的 RAM，下文姑且称其为 NVRAM)来记录操作日志，只要日志被成功存入了这个 RAM，就可以立即通知上层写入成功。

一定要搞清楚日志和数据缓存的区别。日志只是记录一种操作动作以及数据内容，而不是实际数据块，前者比后者要小很多。实际数据块保存在 RAM 中而不是 NVRAM 中。

由于用 RAM 来保存日志，所以速度超级快，可以一次接收上千条写入请求而直接向上层应用返回成功信号，待 RAM 半满或每 10 秒钟的时候，这些数据由 WAFL 一次性批量连续写入硬盘，保证了高效率。这也是 NetApp 的 NAS 为什么相对比较快的一个原因。

10.6.4 WAFL 从不覆写数据

每当 NVRAM 中的日志占用了整个 NVRAM 空间的一半或每 10 秒钟也可能是其他的某些条件达到临界值的时候，WAFL 便会将所有缓存在内存缓冲区内的已经改写的数据以及元数据批量写入硬盘，同时清空操作日志，腾出空间给接下来的请求使用。这个动作叫做 CheckPoint。

WAFL 并不会覆盖掉对应区块中原来的数据，而是寻找磁盘上的空闲块来存放被更改的块。也就是说，所有由 WAFL 写入的数据都会写入空闲块，而不是覆盖旧块。另外，在 CheckPoint 没有发生的时候，或者数据没有全部被 Flush 之前，WAFL 从来不会写入任何元数据到磁盘。

有了以上两个机制就可以保证在 CheckPoint 没发生之前，磁盘上的元数据所对应的实际数据，仍然为上一个 CheckPoint 时候的状态。如果此时发生突然断电等故障，虽然可能有新数据已经被写入空闲块，但是元数据并未写入，所以磁盘上保存的元数据还是指向旧块(这也是为何旧块从来不会被 WAFL 覆盖的原因)，数据就像没有变化一样，根本不用执行文件系统检查这种耗时费力的工作。一旦 CheckPoint 被触发，则 WAFL 先将缓存中的所有数据写入磁盘空闲块，最后才将元数据写入硬盘。一旦新的元数据写入了硬盘，则新元数据的指针均指到了方才被写入的新数据块，对应的旧数据块则变为空闲块(虽然块中仍有数据，但是已经没有任何指针指向它)。

这个特性使得 NetApp 的快照技术水到渠成，且性能良好。

10.7 初露锋芒——BlueArc 公司的 NAS 产品

上文中所述的 IT 系统架构发展的第八阶段(收缩阶段)，其代表就是软件的全芯片化。这些芯片不同于 CPU，而是完全的应用逻辑芯片，比如 ASIC 或更高成本的 FPGA 和 CPLD。这些芯片对于专用逻辑的运算速度远高于 CPU，因为 CPU 是读取外部程序代码指令流来执行并生成结果的，而专用芯片则是通过读入原始数据信号，在经过内部逻辑电路之后直接生成了输出信号。一片频率 100MHz 的 FPGA 在运行专用逻辑的时候速度也会高于频率几 GHz 的 CPU。图 10.45 所示的为 CPU 的处理流程，而图 10.46 所示的是 FPGA 的处理流程。

FPGA 芯片的频率目前已经可以超越 1GHz，其内部电路也已经可以达到 1000 万门。其可容纳的逻辑更多更复杂，处理速度越来越快。

目前有些厂家正在尝试利用 FPGA 来取代 CPU。由于 FPGA 可重构计算的特性，人们认识到许多能发挥其特长的应用。比如在玩《Crysis》电脑游戏时，可将 FPGA 配置成 128 位的高性能 3D 图像处理器；当需要听高保真环绕立体声时，可将 FPGA 配置成专用的 DSP 处理器；在高层网络交换机需要支持新协议时，只需重新配置 FPGA 而不必更改任何硬件；在数字电视变更解码协议时，只要通过网络下载数据来重新配置 FPGA；当从 GSM 网转到

CDMA 网时，也只需重新配置 FPGA 而不必更换手机了。

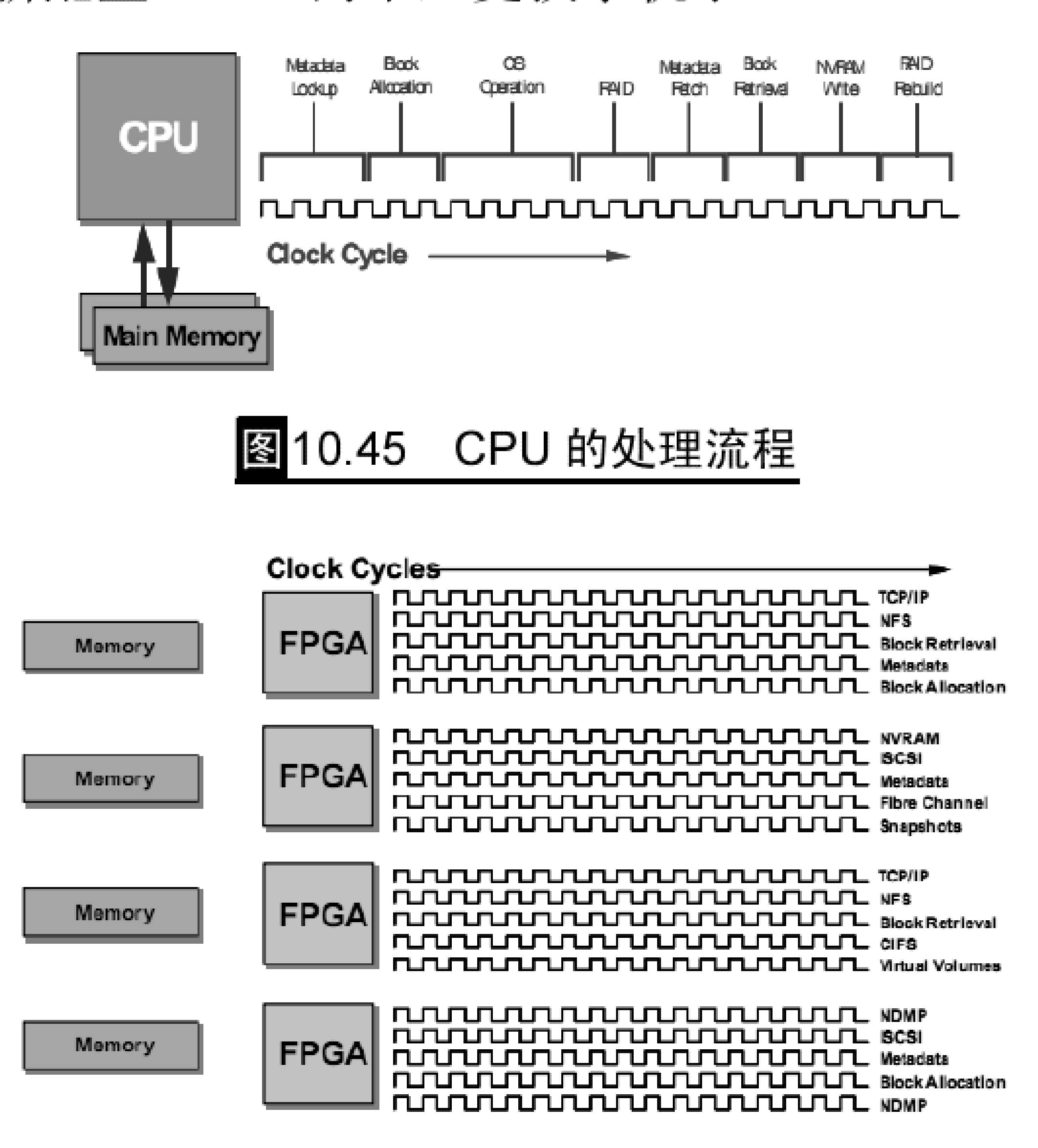


图10.45　CPU 的处理流程

图10.46　FPGA 处理流程

同样，存储产品公司 BlueArc 在其 NAS 产品 Titan 系列中，将其上运行的所有软件逻辑都写入了 FPGA 中。其产品将存储系统路径上的多个模块也分别做成了可插拔式的模块，包括前端网络接口模块、文件系统模块和后端网络接口模块。前端接口是面对客户端的接口，文件系统则是整个系统的处理中枢，后端接口则是连接磁盘扩展柜的接口。每个模块上均有多个 FPGA 芯片来处理各自的逻辑。图 10.47 显示了 Titan 各个模块之间的生态架构图。

图 10.48 为 Titan 产品实物图。后面有四个插槽模块，中间两个为文件系统模块，最上面和最下面的模块分别为前端网络模块和后端网络模块。

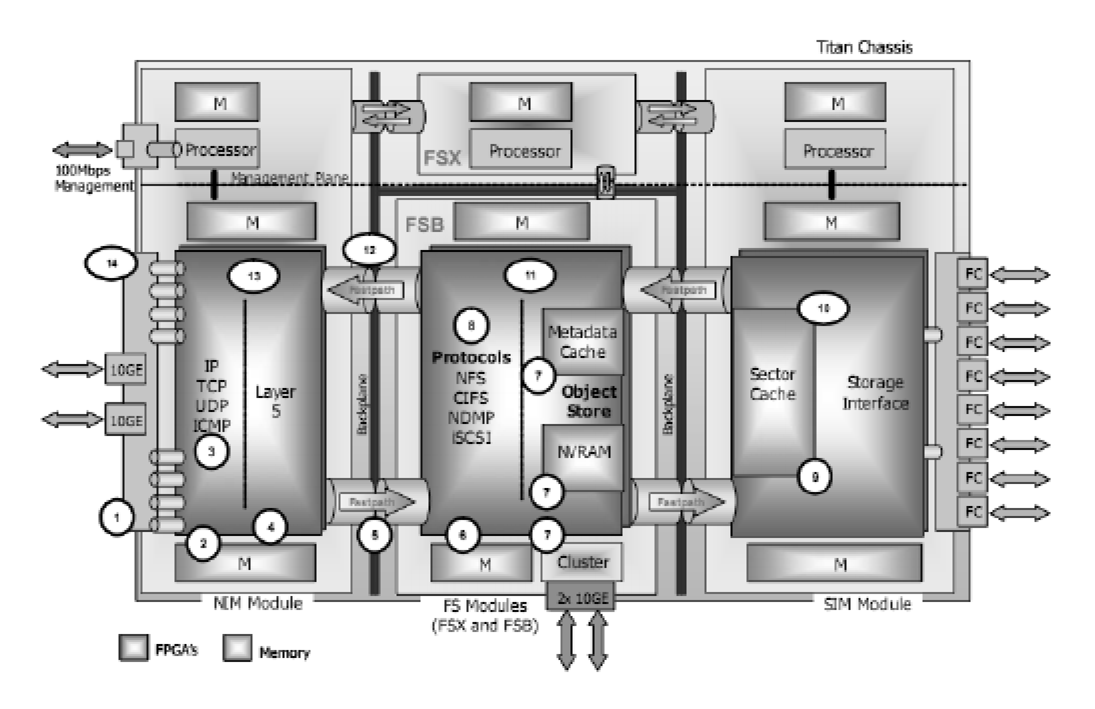


图10.47　Titan 存储产品内部架构示意图



图10.48　Titan 存储产品实物图

且不说 Titan 这种全硬件架构是否成熟，其内部软件是否兼容性良好，抛开这些因素不谈，这种架构其实反应的是一种趋势，一种精神。

读 书 笔 记



第十一章 大师之作

大话以太网和 TCP/IP 协议

- 以太网
- IP
- TCP
- UDP
- OSI

在老祖宗的 OSI 宝典的基础上，后人不知哪位大侠，创立了自己的功夫秘笈，后广泛流传于江湖。这套秘笈，就是大名鼎鼎的“以太网和 TCP/IP 大法”。

11.1 共享总线式以太网

如何让多台 PC 之间能够相互连接并且相互发送信息呢？既然说到独立系统之间的互联，那就可以参考 OSI 来做。在参考 OSI 之前先参考三元素，即连起来、找目标、发数据。

11.1.1 连起来

“连”这个动作，包括了 OSI 的物理层和数据链路层。首先要找一种连接方式将所有节点连接起来。连接多个节点最简单的办法就是总线技术。总线就是一个公共的媒介。要交流，就必须提供交流所需的场所，这个场所就是总线。将总线想象成一根铜导线，每个节点都连接到这条线上。这样每个节点的信号，在总线上所有的其他节点都会感知到，因为良导体上的电位处处相等。

基于这个总线模型，早期的以太网是使用集线器(HUB)将每台 PC 都连接到它的一个接口上，所有这些接口通过集线器里面的中继电路连接在一起。为什么需要中继器，为什么不可以直接物理连接在一起呢？使用中继器的主要原因有两个。第一，信号在总线上传输时受到干扰可能会迅速衰减。加了中继器后集线器将从一个接口收到的 bit 流复制到每个接口，这样就避免了信号衰减。第二，中继器可以防止由于不可知的原因，造成两个节点同时向总线上放置信号所造成的短路。图 11.1 所示的就是一个 HUB 模型。

在数据链路层，以太网使用帧的形式来发送数据流。上层的数据流被封装成一个个的以太网帧，在总线上传播。

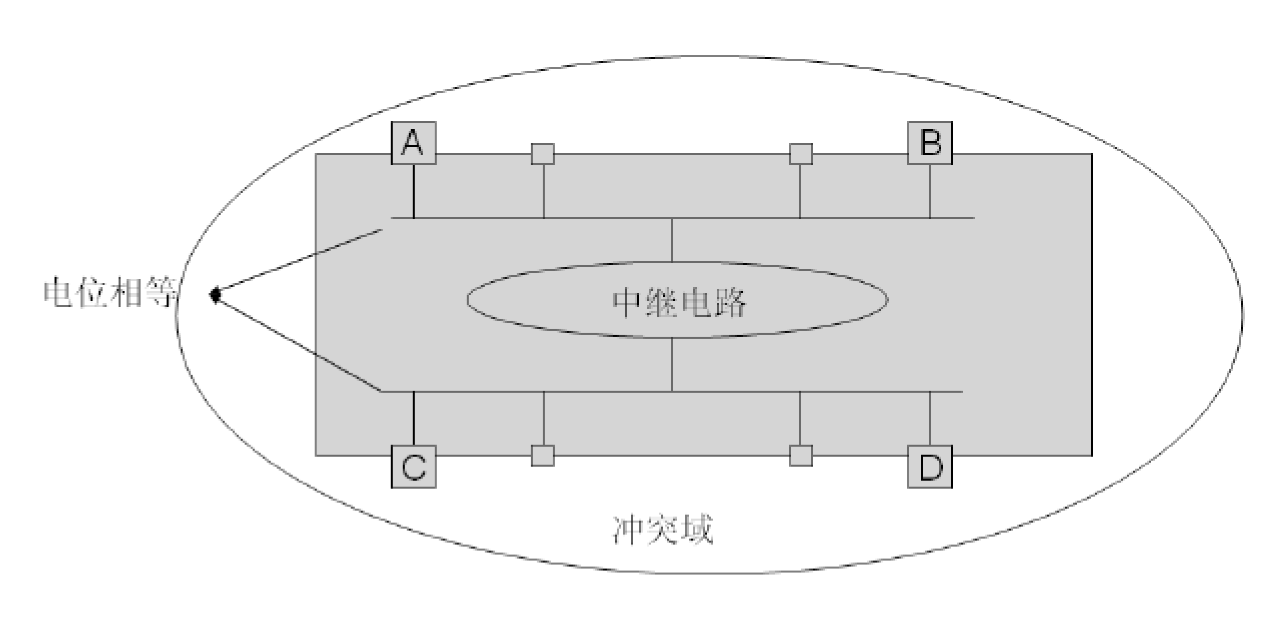


图11.1 HUB 模型

11.1.2 找目标

为了区分总线上的每个节点，每个节点都必须具有一个唯一的身份标志。以太网中，这个标志被称作“Media Access Control(MAC)”地址，介质访问地址，即只有数据帧中包含这个地址，总线介质上拥有这个地址的接收方才知道这个数据帧是给自己的，从而才会将其保存到缓冲区内。实际上，每个以太网帧中都包含源 MAC 地址和目的 MAC 地址。

MAC 地址是一个 6 字节(48bit)长的字段，每个节点的网卡都有一个全球唯一的 MAC 地址，这个地址在网卡出厂时候被固化在芯片中。

以太网就是利用 MAC 地址来区分每个节点的。

11.1.3　发数据

既然是总线方式联网，那么每个节点发出的信号，总线上的所有节点都会感知到，并且，同一时刻只能由一个节点的信号在总线上传递，如果同时有多个节点都向总线上传递信号，则各路信号之间就会发生冲突，比如，节点 A 在时间点 T1 时刻向总线上放置高电位信号(假设电位为 0.5V)，而节点 B 在 T1 时刻向总线上放置低电位信号(假设电位为 0V)，这样，就形成了电流通路，一方面造成短路，缩短设备寿命，另一方面总线上的其他节点所感知到的电位总是 0。所以，任何情况下，都不能让多个节点同时向总线上放置信号。

有如下两种措施可以防止这种情况的发生。

- 集线器中的中继电路，会防止由于恶意破坏或者其他不可知的程序 bug 所导致的信号冲突。
- 在协议角度，从根本上杜绝这种情况的发生。

在总线上，每个节点利用载波侦听机制(CSMA)来检测当前总线上是否有其他节点的信号正在传播，一旦检测到信号，则暂时不发送缓冲区内的数据帧，并不断地侦听电路上的信号，一旦发现总线空闲，则立即向总线上放置信号，声明要使用总线，如果在完全相同的时刻，另一个节点也同样放置了信号，则两路信号会发生冲突，两个节点检测到冲突后，会撤销声明，继续回到侦听状态，这个过程叫做冲突检测(CD)。

但两个节点在同一时刻同时发出信号的几率很小，即使本轮声明失败，在下一轮争抢声明中，某个节点胜利的几率是很大的，而且以太网中的所有节点的优先权都是一样的，或者可以说以太网内没有优先权的概念，包括网关设备在内。而 SCSI 总线的优先级最高，因为 SCSI 协议本身就是一个 Poll－Response 型的协议，SCSI 控制器要顺序寻找总线上的除自身之外的所有其他节点，它的优先级最高。所以对于以太网来说，完全不必担心某个节点永远也抢不到总线使用权(俗称“总线饿死”)的情况。

另外，也不必担心这种 CSMA/CD 机制会耗费过多的时间，由于这种机制不是靠 CPU 执行代码来实现的，而完全是靠电路逻辑执行的，所以速度都在微秒级，宏观上不会感觉到延迟。

如果一条总线上的节点过多，则发生冲突的几率就越大，造成的开销和延迟也越大。另一方面节点越多，每个节点使用总线的平均时间也就会降低，宏观上，也就造成了每节点的可用带宽降低。所以共享总线方式的网络，可接入的节点数量是有限的。

节点取得了总线使用权之后，便开始发送数据帧，也就是将数据帧的 bit 流转换成电信号，以一定的间隔速度放置到总线上，对于 10Mb/s 以太网来说，每隔一千万分之一秒，总线上的信号就被放置一次，直到整个数据帧被传播完毕，总线空闲，然后节点再发起新一轮的 CSMA/CD 过程。

在一个节点向总线上传播数据的同时，所有其他节点都会将总线上正在传播的信号保存到各自的缓冲区内，形成一帧一帧的数据。也就是说，共享式以太网中，任何一个节点所发送的数据，其他所有节点都会收到。

这岂不是没有隐私可言了么？

的确，只要在 HUB 上的任何一个节点上安装一个抓包软件，就可以抓取总线上的信号，

看到任何源节点发送到任何目的节点的数据。

比如，节点 A 想与节点 B 通信，节点 B 怎么知道当前总线上的数据是给它的呢？我们上面讲过，每个节点都有一个 MAC 地址，节点 A 若想将数据传送给节点 B，就会在数据帧中的特定字段填入节点 B 的 MAC 地址(目的 MAC 地址)，并且在特定字段填入节点 A 自己的 MAC 地址(源 MAC 地址)，这个数据包被所有人收到，当然包括 B 节点。B 节点检测这个帧的目的 MAC 地址，与自身的 MAC 地址做对比，如果相配，则得知这个帧是给自己的，并且通过检测源 MAC 地址段，B 会知道这个帧是节点 A 给的。而节点 C 此时也收到了这个帧，它也会对比自己的 MAC 与帧中的目的 MAC 字段，发现不匹配，所以知道这个帧不是给自己的，立即将其丢弃。

以太网定义了一种特殊的 MAC 地址，这个地址的所有 48 位的值都为 1。这个地址对应了总线上的所有节点，也就是说，如果某个数据帧中的目的地址是这个特殊的 MAC 地址，则所有接收到这个帧的节点都会保存起来并提交上层协议处理，而被这个地址称为广播地址。

11.2　网桥式以太网

前面描述了共享总线型以太网。会发现，一个人说话所有人都听到，这不安全，也没必要浪费资源。必须发明一个机制来改进，网桥的出现，初步降低了这个问题所带来的影响。

网桥也有多个接口，外观甚至和 HUB 一样，但是里面的电路加入了逻辑运算电路和智能化的东西，不像 HUB 一样，所有接口复制总线上的数据。

网桥把它上面的接口分了很多组，每一个组中的接口都在一条独立的总线上，不同的组使用不同的总线。也就是说，除非网桥转发过去，否则某个组中总线上的数据，不会被另外的组收到。这样就减小了冲突域，相当于本来 10 个人的小组，分成了多个子讨论组，一个讨论组在讨论的同时，另一个讨论组也可以讨论，两个组完全物理上隔离。这样，接入相同数目的节点，但是性能却比 HUB 好多了。图 11.2 为一个网桥模型。

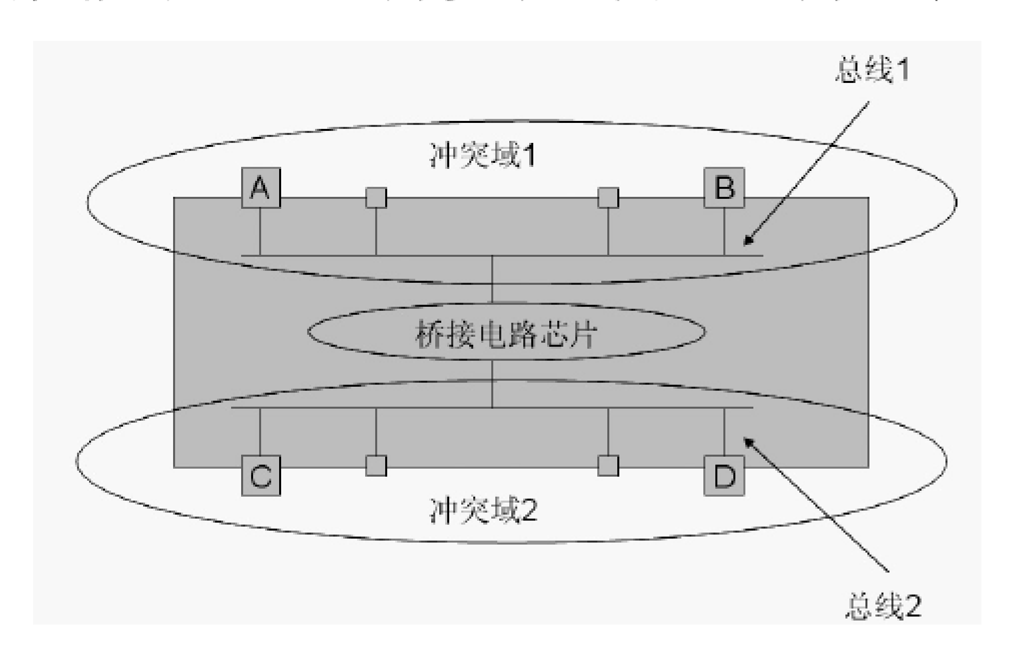


图11.2　网桥模型

如图 11.2 所示，节点 A 与 B 在总线 1(冲突域 1)中，总线 1 与总线 2 是隔离的，如果 A 与 B 通信，则只会在总线 1 上冲突，A 与 B 通信的同时，总线 2 上的 C 与 D 也可以通信。这样，每个冲突域中所包含的节点数目降低了，所以相对 HUB 来说提高了性能。

但是如果 A 想与 D 通信，怎么办呢？毫无疑问，桥接芯片必须将 A 发出的数据帧从总

线 1 复制到总线 2 上，反之亦然。那么，桥接芯片如何知道某个数据帧是 A 发送给 D 的，而不是 B 的呢(A 发送给 B 的数据帧不能被复制到总线 2，否则就和 HUB 一样了)？显然，桥接芯片必须对数据帧做分析，分析其中的目的 MAC 地址，如果某个数据帧的源节点在总线 1 上，其目的 MAC 地址对应的节点也在总线 1 上，那么芯片就不会复制这个数据帧到其他总线，但是如果目的 MAC 对应的节点与源节点不在一个总线上，则桥接芯片会将这个数据帧转发到目的节点所在的总线，从而让目的节点收到。

为了做到这一点，网桥必须知道某条总线上到底包含了哪些 MAC 地址，这样才能做到根据目的 MAC 与总线的对应关系来决定某个数据帧是否需要转发，转发到哪条总线。桥接芯片是通过动态学习来填充“MAC－总线”对应表的。

每当总线上有数据帧通过，桥接芯片就去提取它的源 MAC 地址，放入对应表中(已经存在的记录将会忽略)。比如，当网桥刚刚加电的时候，这张对应表是空的。这时总线 1 上的 A 向 B 发送了一个数据帧，因为是共享总线，所以桥接芯片也会同样收到这个帧，网桥立即提取这个帧中的源 MAC 地址，假如这个 MAC 值为：00-40-D0-A0-A3-7D，就向表中填入一条“00-40-D0-A0-A3-7D   bus1”，这表示 bus1 总线上，有一个 MAC 地址为 00-40-D0-A0-A3-7D 的节点连接着。同样，所有总线上所收到的源 MAC 地址都会被记录到表中。如果从总线 1 上收到一个数据帧，经过查表发现其目的 MAC 在另一条总线上，则网桥将这个帧转发到那条总线。如果某个帧的目的 MAC 还没有被网桥学习到，也就是说网桥不知道这个 MAC 在哪个总线，则网桥便向除发送这个帧的总线之外的其他总线转发这个帧，直到学习到这个 MAC 条目。



网桥并不记录到底是哪个端口所连接的节点具有这个 MAC，因为根本没有必要记录端口信息，数据帧只要进入这条总线，这条总线上所有端口下连接的节点就都会收到数据，只记录“总线－MAC”统计表即可，所以没有必要记录“端口－MAC”表。

11.3 交换式以太网

HUB 是一锅粥，网桥也好不到哪里去，只不过把这一锅大粥分成了多锅小粥而已。

随着硬件技术的提高，交换式以太网出现了。交换式以太网利用交换机替代了网桥。其实交换机也是一种网桥，一种特殊的网桥。普通网桥中，每个端口组(冲突域)中有不只一个端口，而交换机中，每个端口组中只有一个端口，也可以这么说，交换机上的每个端口都独占一条总线。这样交换机彻底隔离了冲突域，而不是仅仅减小了冲突域，也就是说，交换机的每个端口下都是一个独立的冲突域。

此外，交换机内部芯片，也不再被称作桥接芯片，而改成交换芯片。网桥芯片学习并记录“MAC－总线”对应表，那么顺藤摸瓜，交换机既然每个端口独占一个总线，理所当然的，交换芯片学习并记录的就是“MAC－端口”对应表。

交换机是以太网的最终实现形式。端口之间不再是共享总线通信，而是可以在任意时刻、任意两个端口之间同时收发数据。

HUB 是没有大脑的，只有一个中继器。网桥进化出了大脑，但是智商不高。而交换机则拥有高智商的大脑和极快的运算速度。图 11.3 便是一个交换式模型。

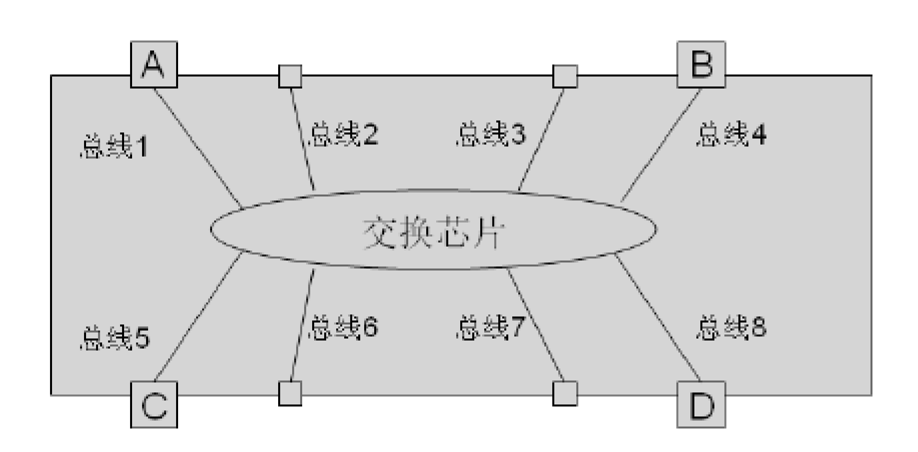


图11.3　交换式以太网模型

11.4　TCP/IP 协议

以太网的出现，给系统间互联提供了方便的方式。每个节点安装一块以太网适配器，上层程序只要将要发送的数据以及数据要到达的目的 MAC 地址告诉以太网卡，数据就可以通过以太网传输到目的，完成通信过程。

但是实际中，以太网并不是直接被应用程序使用来收发数据的。究其原因，就是因为以太网是一个没有传输保障机制的网络。首先以太网设备不会对数据帧进行校验纠错等措施。其次以太网设备一旦由于数据交换量过大，就可能造成缓冲区队列充满而主动丢弃数据帧，而且丢弃数据帧的同时，不会有任何措施来通知发送方。所以，以太网是一个不可靠的网络。应用程序如果直接调用以太网来发送数据，则必须忍受丢失数据的风险。

为了解决以太网的这个弱点，人们在以太网之上，增加了一个层次，在这个层次上，人们实现了以太网所不具有的功能。运行在这个层次上的程序，用了很多复杂的机制来保证发送给以太网适配器的数据包能够成功的到达对方，如果中途发生丢包现象，则本地会通过超时未得到确认等机制来重新发送该数据包。而应用程序由原来的直接调用以太网接口，改为调用这种位于以太网上层的新程序接口从而获得可靠的数据收发。

运行在这个新层次上的协议，有很多，比如 NetBEUI、NetBIOS、IPX、TCP/IP 等。这些协议，其下层调用以太网提供的服务，上层则向外提供新的调用接口，向应用程序提供可靠的网络传输服务。

在这些协议中，TCP/IP 以其广谱的适用性、良好的性能以及能够良好的在超大规模网络上运行等优点，迅速地得到了普及，成为 Internet 网络所使用的通信协议。

11.4.1　TCP/IP 协议中的 IP

到 MAC 这一层，以太网已经实现了 OSI 的下三层，即物理层、链路层和网络层。以太网也只跨越了这 3 层，从第四层到第七层，以太网没有涉足(现在普遍认为以太网只作用到链路层，这是一个错误观点。造成这种误解的原因，是以太网一旦与 TCP/IP 结合，便沦落为 TCP/IP 协议的链路层，其第三层地址被以太网的 IP 所映射掉了，所以掩盖了其第三层的元素)。

由于以太网的天生弱点，使它不得不选择与 TCP/IP 协议合作，求助 TCP/IP 协议向上层应用程序提供可靠传输保障。由于主动权完全掌握在 TCP/IP 手中，所以 TCP/IP 向以太

网提出了一个非常过分的要求，即以太网要想占有市场一席之地，就必须将 MAC 地址隐藏掉，对外统统用 TCP/IP 家族的新一代地址：IP 地址。以太网委曲求全，不得不同意这个要求。

其实，TCP/IP 提出这个要求不是故意难为以太网。因为除了以太网之外，还有很多其他类型的联网方式。而以 TCP/IP 在业界的权威性，其他网络都求助于 TCP/IP 来将它们融入市场以分一杯羹。而几乎每种联网方式都有自己的编址和寻址方式，面对这么多种地址，TCP/IP 只好快刀斩乱麻，将这些五花八门的地址，统统映射到 IP 地址上，对外统一以 IP 地址作为编址方式。

ARP 协议

既然 TCP/IP 宣称要将所有类型的地址全部统一到 IP 地址，那么也就意味着，网络中的每个节点，都必须配备一个 IP 地址，节点之间相互通信的时候，也要使用 IP 地址。但是，数据帧最终是通过底层的网络传输设备来转发的，比如以太网交换机。而以太网交换机是不理解 IP 地址的，它只能分析数据帧中特定偏移处的 MAC 地址，从而做出转发动作。所以，必须要有一种机制，来将 IP 地址映射成底层 MAC 地址。

ARP 协议就是专门用来处理一种地址与另一种地址之间相互映射的一种机制。ARP 协议运行在每个网络设备上，将一种地址映射成底层网络设备所使用的另一种地址。

对于 ARP 协议，本书不再做具体描述。

11.4.2 IP 的另外一个作用

IP 层还有一个作用，就是适配上下层，给链路层和传输层提供适配。适配了什么，怎么适配的呢？我们知道链路层有 MTU 的概念，也就是链路最大传输单元，即没帧所允许包含的做大字节数。



前面第 7 章中描述过，链路层就类似于司机和交通规则，要对货车的载重，车型大小有要求，这条路承受不了多重的车，每辆车不能拉超过多高的货物等规则。如果客户给的货物太大、太重，不能一次运过去，那么只能把货物分割、分次运送，到达目的地后，再组装起来。这个动作是由 IP 层程序来做。也就是 IP 根据链路的 MTU 值来分割货物，然后给每个分割的货物块，贴上源和目的 IP 地址、顺序号，以便在货物块到达目的地后，利用顺序号来重新合并成完整的一件货物。

货物被分割成块后都需要被路由转发一次，然而路由器每次选择的路径不一定都一样，而且每个块都需要由司机运送，司机驾驶水平、速度不同(链路层)，就难免会有些块先到、有些后到，所以到达目的地后很有可能乱序。此时就要用到顺序号了，这个号码是根据货物被分割处相对整个货物起点的距离(offset，偏移量)而制定的。根据这个号，等所有货物块到达目的之后，对方的 IP 程序就会根据这个号码将零散的块组装起来。



每个货物块都携带 IP 头部，但是只有第一块携带 TCP 或者 UDP 头部，因为传输层头部是在应用数据之前的，IP 分割的时候，一定会把传输层头部分割在第一块货物中。

11.4.3　TCP/IP 协议中的 TCP 和 UDP

TCP/IP 协议其实包含了两个亚层，IP 是第一个亚层，也就是用来统一底层网络地址和寻址的亚层。第二个亚层，就是 TCP 或者 UDP，在逻辑上它们位于 IP 之上。

TCP 的功能，就是维护复杂的状态机，保障发送方发出的每个数据包，都会被最终传送到接收方，如果发生严重错误，还会向上层应用反馈出错信息，从而保证应用层逻辑的无误和一致性。

而 UDP 的功能，则可以理解为是 TCP/IP 对以太网的一种透传，即 UDP 是一个没有传输保障功能的亚层，除了 UDP 可以提供比以太网更方便的调用方式外，其他方面没有什么本质区别。也许是因为 TCP/IP 协议觉得 TCP 的逻辑太过复杂，所以提供了一种绕过 TCP 复杂逻辑而又比以太网更加方便调用的方式，即 UDP。

TCP/IP 协议向上层应用程序提供的调用接口称为 Socket 接口，即“插座”接口。这也体现了 TCP/IP 想让应用程序更为方便的使用网络，就像将插头插入供电插座而接入电网一样使用计算机网络。

基于 TCP/IP 有很多应用层协议，这些协议必须依赖 TCP/IP 协议，比如：Ping、Trace、SNMP、Telnet、SMTP、FTP、HTTP 等。这些应用程序，加上它们所依赖的 IP、TCP 和 UDP，然后加上物理层链路层(以太网等)，一并形成了 TCP/IP 协议簇。

1. TCP 协议

第 7 章中说过，TCP 就是一个押运员的角色，也就是由它把货物交给 IP 做调度的。货物最初是由应用程序来生成的，应用程序又调用 Socket 接口来向接收方发送这些货物。应用程序通告 TCP/IP 去特定的内存区域将数据拷贝到 Socket 的缓冲区，然后 TCP 再从缓冲区将数据通过 IP 层的分片后，从底层网络适配器发送到网络对端。

TCP 通过 MSS(Max Segment Size)来调整每次转给 IP 层的数据大小。而 MSS 的值完全取决于底层链路的 MTU 值。为了避免 IP 分片，MSS 总是等于 MTU 值减掉 IP 头，再减掉 TCP 头之后的值。这样 TCP 发送给 IP 的数据，IP 加入 IP 头之后，恰好就等于底层链路的 MTU 值，使得 IP 不需要分片。



既然货物的大小要匹配 MTU，那为什么不直接让 TCP 直接把分割好的货物给 IP 呢？

TCP 其实很想这么做，但是它很难做到，因为 TCP 是个端到端的协议，也就是说，只有通信的最终端点维护着 TCP 状态信息，途经的各个其他设备一概不知道。如果两个端点所处的局域网都是以太网，但是途经一段串口链路，假设串口 MTU 为 576 字节，以太网 MTU 为 1500 字节，那么双方在 TCP 握手的时候会互相通告自己的 MSS，因为是端到端协

议，不关心途经的设备和链路，那么他们都认为自己和对方都处在以太网中，所以互相都通告自己的 MSS 值为 1460 字节(最大分段大小，等于出口 MTU 减去 IP 头和 TCP 头的开销)。

这样，TCP 给 IP 的货物大小就是 1460+TCP 头=1480 字节，然后加上 IP 头传输出去。一旦这个数据包到达了串口链路，则串口链路两端的 IP 层必须根据串口链路的 MTU，对(1480+串行链路协议头)字节的数据包进行分片，将数据包分成多个 576 字节的数据帧(最后一帧大小可能小于 576 字节)，从而在串口链路上传输。所以 TCP 的 MSS，在广域网传输时基本派不上用场。不过，有机制可以探测到途经链路上的最小 MTU 值，TCP 参考这个链路最小 MTU 值所得出的 MSS 值，这时是很有价值的。

在 Windows 系统中，每块网卡的 MTU 大小其实是可调的，但是只能调节到比网卡所连接到的以太网交换设备所允许的最小 MTU 值还小才可以，如果调节到大于这个值，则会造成数据丢失以及不可知的莫名错误。Windows 中是通过注册表中以下的键值来调节 MTU 大小的：

```
HKEY_LOCAL_MACHINE\SYSTEM\ControlSet003\Services\TCP/IP\Parameters\Inter
faces\{接口编号}\MTU
```

图 11.4 是一个 TCP 握手过程中，发起连接端在 TCP 头的 option 字段中给出了本地 TCP 的最大 MSS 值(本地网络适配器最大 MTU 值减去 40 字节)。本例中由于 MTU 值被配置为了 1300 字节而不是默认的 1500 字节，所以造成 TCP 通告 MSS 值为 1260 字节。

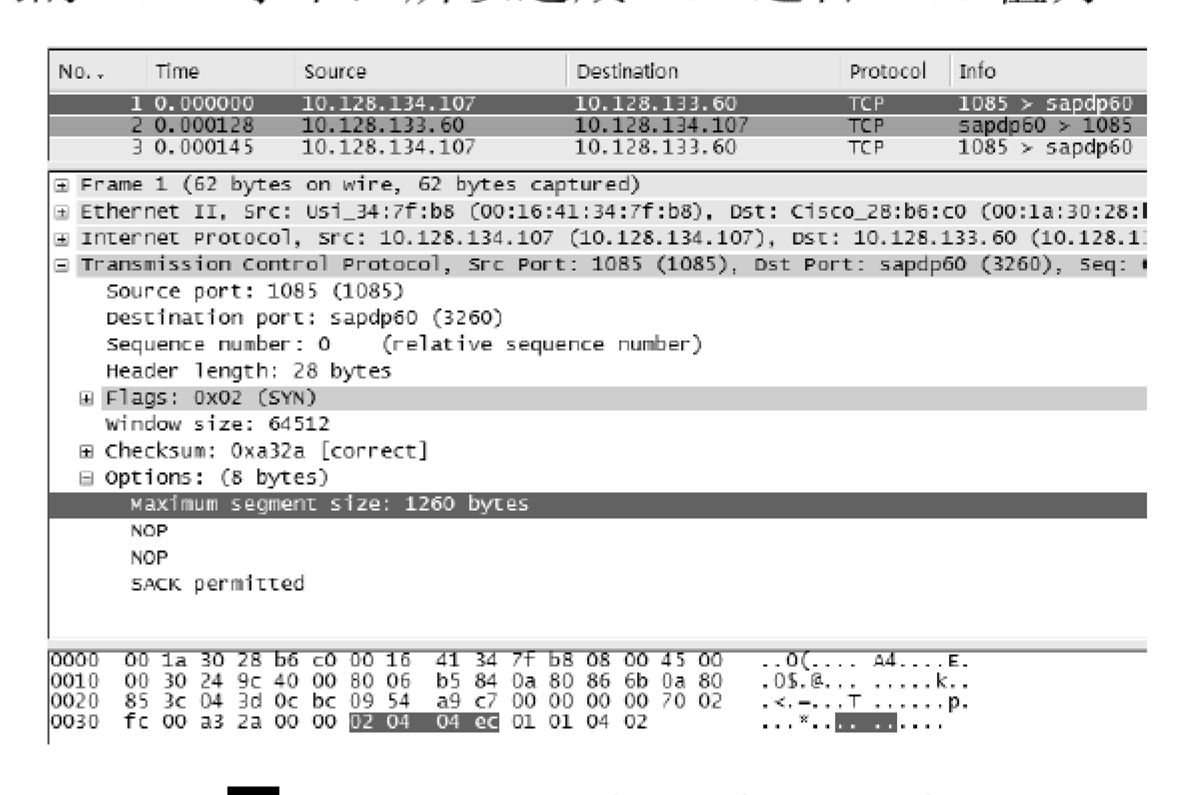


图11.4 TCP 握手过程中的一个包

TCP 将上层应用的数据完全当成字节流，不对其进行定界处理。TCP 认为上层应用数据就是一连串的字节，它不认识字节的具体意义，却可以任意分割这些字节，封装成货物进行传送，但是必须保证数据的排列顺序。

比如，上层应用要传输 123456789 这 9 个数字，TCP 可以一次发送这 9 个字节的内容，也可以每次发送 3 个字节，分 3 次发完，或者，上层放到货仓中 1、2、3 三个数据，TCP 有可能将这 3 个数据打包一次发送(nagel 算法)，TCP 的这些动作，上层都是不知道的。但是，对于 123456789 这 9 个数字，如果接收端先收到 1、2、3 这三个数字，4～9 还没有被接收到，而当只有应用程序完全接收到这 9 个数字才会认为数据有意义时，应该怎么办呢？

要解决这个问题，就得完全靠上层应用程序。比如，可以在一个消息头部增加一个定长字段，表示这个消息的长度，比如在 123456789 之前增加一个(9)，变成(9)123456789，那么程序一旦接受到(9)，就能判断出接下来要连续接收 9 个字节才有意义，如果某时刻只

接收到了 1、2、3，应用程序便会将数据缓存起来，然后等待后面的 6 个数字。

如果 TCP 从(9)处分割了数据流，怎么办？比如分割成“(”和“9)123456789”两段数据，那么此时接收方应用程序接收到“(”这个字符，就会不理解其意义，这样就要求应用程序必须有缓存，把 TCP 交上来的数据流放到缓存中，然后自行合并成有意义的数据后再做处理。

TCP 把上层的数据看作一些无关联逻辑的数据流，它不会感知到消息与消息之间的定界符。消息和消息之间的界限需要完全由应用程序自行分析。接收端和发送端的 TCP 都保存一个缓冲区(货仓)，发送和接收的数据都存放在货仓中，接收方 TCP 货仓中的数据，每次被应用程序一次性取出到应用程序自己的缓冲区，应用程序再从应用缓冲中将数据流连接成有意义的数据进行处理。

一句话，TCP 是把上层数据“分段”，IP 是把 TCP 分好的段再“分片”(如果这个段大于 MTU)，IP 到达目的之后会把每个分片合并成一个 TCP 的“分段”，提交给 TCP，然后 TCP 就直接存放到货仓，顺序排放，不管上层消息间的分界。

2. UDP 协议

UDP 和 TCP 不同，TCP 是孙悟空，那么 UDP 就是猪八戒。UDP 只是被动的起到一个 IP 和上三层之间的接口作用，UDP 没有传输保障机制，出错后不会重传，不需要保持重传缓冲和复杂的定时器、状态机等机制，而且 UDP 也不会像 TCP 那样把数据流按照 MSS 分段，UDP 统统不理会。用户传给 UDP 多大的数据，它就一次性发送出去，适配 MTU 的工作完全由 IP 来做。

UDP 没有握手机制，想发就发，发完就不管了。正因为 UDP 这么简单，所以 UDP 头部只有 8 字节长，包括目的和源端口号、UDP 数据包长度、UDP 校验和。而且 UDP 相对 TCP 效率高了很多。所以它适合用在实时性要求很高，但是可靠性要求不高的时候，比如实时视频流、音频流服务等。

3. 端口号

计算机操作系统上，运行着 N 个程序，也可以说是 N 个进程。如果程序之间需要相互通信，就需要用号码来标识各个程序，也就有了进程号的概念。只要知道一个程序的进程号，就可以用这个区分其他程序。

同一台计算机上程序之间的通信，一般是在内存中直接通信的。如果两台计算机上的两个程序之间需要通信，虽然也可以通过高速网络将两台计算机的内存共享，但是这种网络的成本很高。而普通网络能做到的，只能用另外一种方式，即先将消息通过网络发送到另一个计算机，然后让接收到消息的计算机来选择把这个消息发给其上运行的对应的程序。

要这样做，接收方的操作系统中的 TCP/IP 协议就一定需要知道某个数据包应该放入哪个应用程序的缓冲区，因为同一时刻可能有多个应用调用 Socket 进行数据收发操作。为了区分开正在调用 Socket 的不同应用程序，TCP/IP 协议规定了端口号的概念。任何一个应用程序在调用 Socket 的时候，必须声明连接目的计算机上 TCP/IP 协议的那个端口号。

11.5 TCP/IP 和以太网的关系

提示 很多人把 TCP/IP 和以太网硬性关联起来，认为 TCP/IP 就是以太网，或者以太网就是 TCP/IP，这种思想是完全错误的。

TCP/IP 是一套协议体系，以太网也是一套协议体系，他们之间是相互利用的关系，而不是相互依存的关系。

TCP/IP 协议并不像以太网一样有其底层专门的硬件，但是它可以租用一切合适的硬件来为它充当物理层和链路层的角色。除了以太网交换机，TCP/IP 甚至可以用无线电波、红外线、USB、COM 串口、ATM 等作为其物理层和链路层。

以太网给 TCP/IP 充当了链路层，不一定代表它只能作用于链路层。我们知道以太网有自己的网络层编址和寻址机制，它有网络层的元素。各种联网协议都有自己的层次，都在 OSI 模型中有自己的定义，只不过 TCP/IP 协议，它在网络层和传输层的功能应用的太广泛了，所以 OSI 的第三层和第四层，几乎就是被 TCP/IP 协议给统治了，其他协议虽然也占有一席之地，但是相比 TCP/IP 的光辉就暗淡了许多。另外，TCP/IP 没有统治链路层和物理层，在这两层中，就是其他协议体系的天下，所以 TCP/IP 只能“租用”其他底层协议，比如以太网，来完成 OSI 开放系统互联的任务。

这就是 PoP，意即 Protocol over Protocol。PoP 的思想，到处可见，因为没有人可以统治 OSI 的全部 7 层，毕竟需要大家相互合作。关于 PoP 的具体分析，在本书后面的章节中会讲到。

读 书 笔 记



存储网络的新军 IP SAN

- TCP/IP
- 以太网
- SAN
- PoP

TCP/IP 协议可谓出尽了风头，不仅统治了 Internet，就连局域网通信，人们也愿意使用高开销的 TCP/IP 协议，可算是给足了它面子。

TCP/IP 的买卖越开越大，吞并收购，不断涉足新领域，甚至连家用电器都想接入 IP 网络。而偏偏有一位愣是坚持不给 TCP/IP 面子，这就是大名鼎鼎的 FC 大侠。十几年来，TCP/IP 在江湖上可谓是叱诧风云、前呼后拥、一呼百应，听惯了恭维话，看惯了鞠躬人。但是唯独 FC 大侠从来没正眼看过它一次，TCP/IP 心里窝火啊！

12.1 横眉冷对——TCP/IP 与 FC

FC 大侠既然敢和 TCP/IP 叫板，肯定有自己的拿手本领。

- 首先，FC 在家底儿上就占了上风，FC 是正儿八经的世家，四世同堂(OSI 下四层都有定义)。而 TCP/IP 就两辈人(网络层和传输层)，好不容易整了个后代还是收养的(租用以太网等其他底层传输网络)。
- 其次，FC 目前普遍能跑出每秒 4Gb/s 的速度，以太网每秒 100Mb/s 的速度算正常，1Gb/s 的速度算超常发挥，10Gb/s 的速度还正在修炼之中。
- 再次，TCP/IP 办事拖泥带水，笨重不堪。瞧瞧 TCP/IP 的那个大头(TCP 和 IP 头开销合起来要 40 字节)，下雨都能当伞用。FC 的脑袋只有 24 字节。以太网交换机 MTU 一般为 1500 字节，FC 交换机则超过 2000 字节，传输效率上也高过以太网。

正由于这些原因，服务器和存储都愿意走 FC 的道儿。

老 T(TCP/IP)心里一琢磨，虽然俺家业不如 FC 大，跑的不如它快，长得也比它胖，但俺也不是一无是处啊！

- 首先，俺广结天下良友，江湖各处都有俺的分号。
- 其次，俺便宜，给钱就让走。且俺的好兄弟以太网，几乎有网络的地方，一定少不了它。
- 再次，那 FC 也不是神仙，走它的道毛病太多，兼容性差，扩展性差，而且费用太高。就凭这三点，不信斗不过 FC。

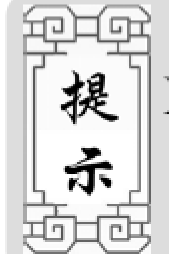
老 T 陷入了久久的沉思之中……

12.2 自叹不如——为何不是以太网+TCP/IP

以太网可寻址容量很大，甚至比 IP 的地址容量都要大，是 IP 的 2 的 16 次方倍。而其地址是定长的，且使用专用电路完成交换动作。以太网除了双绞线之外，还可以用光纤进行传输。最重要的一点就是以太网非常廉价、部署简单。一个普通 16 口 100Mb 以太网交换机，只需要一两百元左右。而一个 16 口的 FC 交换机得上万元，还没有算上适配光纤的 SFP 适配器的费用。

但是，以太网与 FC 网络比起来，也有其先天不足之处。

- 第一，速度方面，以太网目前只普及到 1Gb/s 的速度，虽然 10Gb/s 以太网络已经开发出了成品，但是离完全普及还需要一段时间。而 FC 已经普及到了 4Gb/s 的速度，且 8Gb/s 和 10Gb/s 速度的 FC 接口标准也正在制定当中。
- 第二，以太网是一个不可靠的网络，它不是一个端到端的协议，不管源和目的的状态，只是一味地向接口上塞数据，这也是下层协议的普遍特点。即使对方缓冲将满，以太网还是照样往链路上塞数据，而不会有所减慢。一旦接收方缓存充满，随后的数据帧就会被自动丢弃而不会向上层通告。所以，以太网必须依靠一种提供可靠传输机制的上层协议才能达到可靠传输。


Disk SAN 是唯一一个没有被以太网攻克的领域。最大的原因其实就是因为以太网的速度相对 FC 来说慢了太多。

以太网仿佛总是慢了一步。FC 速率普及到 1Gb/s 的时候，以太网才刚刚普及到 10Mb/s 的速率，而且还是 HUB 总线式以太网。而当 FC 普及到 2Gb/s 速率的时候，以太网也刚刚普及到 100Mb/s。直到目前，以太网才普及到 1Gb/s 速率，而 FC 已经普及到 4Gb/s 了。

以太网即使是依靠了 TCP/IP 协议提供的传输保障机制，也难敌 FC 协议。总之，目前来说，FC 协议在性能方面处处比 TCP/IP+以太网强。

12.3 天生我才必有用——攻陷 Disk SAN 阵地

老 T 深知，自己再怎么修炼，也不可能在速度和性能上与 FC 正面交锋。在江湖中摸爬滚打了这么多年，老谋深算的老 T 决定避开自己的这些短处不谈，发扬自己的长处，将 Disk SAN(相对于 NAS SAN)这个阵地攻陷。

俺老 T 不管是论品相还是论才能，都不输给 FC。既然 SCSI 能嫁给 FC，它就没有理由对俺不动心。老 T 开始做白日梦，憧憬着与 SCSI 成亲后的种种。

老 T 经过一段时间的摸索和实验，终于设计出了一套新协议系统，称其为 ISCSI，即 Internet Small Computer System Interface。即在这种协议中，SCSI 语言甚至可以通过 Internet 来传递，也就是承载于 TCP/IP 之上。

由此可见其扩展性是非常高的。只要 IP 可达，则两个节点之间就可以通过 ISCSI 通信。也就是说，位于中国的一台主机，可以通过 ISCSI 协议从 Internet 访问国外的存储空间。既然 ISCSI 协议是利用 TCP/IP 协议来传输 SCSI 语言指令，那么在通信的双方就一定需要先建立起 TCP 的连接。与 FC 协议类似，ISCSI 将发起通信的一方称为 Initiator，将被连接端称为 Target。一般来说，Initiator 端均为主机设备，Target 端均为提供存储空间的设备，比如磁盘阵列。

老 T 拿着设计蓝图找到了 SCSI 协议，并且成功取得了 SCSI 的芳心。于 2004 年 4 月份完婚，并且领到了结婚证，编号为 RFC3720。

两人并肩携手，成功游说了一批磁盘阵列生产厂商在其产品上尝试着实现 ISCSI 协议。


正如本书第 10 章中所说的，既然 SCSI 语言及数据可以用 FC 协议传递，文件系统语言可以用以太网传递，那么 SCSI 语言当然也可以用以太网传递。

12.4 ISCSI 交互过程简析

以下所有实例均为 Windows XP 操作系统环境，抓包软件为 WireShark 0.99。

12.4.1 实例一：初始化磁盘过程

图 12.1～图 12.6 所示的 Trace 结果是在 Windows XP 中，初始化一块通过 ISCSI 协议提交上来的 LUN 磁盘的过程中抓取的。在 Windows 初始化一块新磁盘的过程中，会对磁盘进行查询以及修改其 MBR。下面就来分析一下具体的动作。

图 12.1 中包含了 Frame1～Frame8。

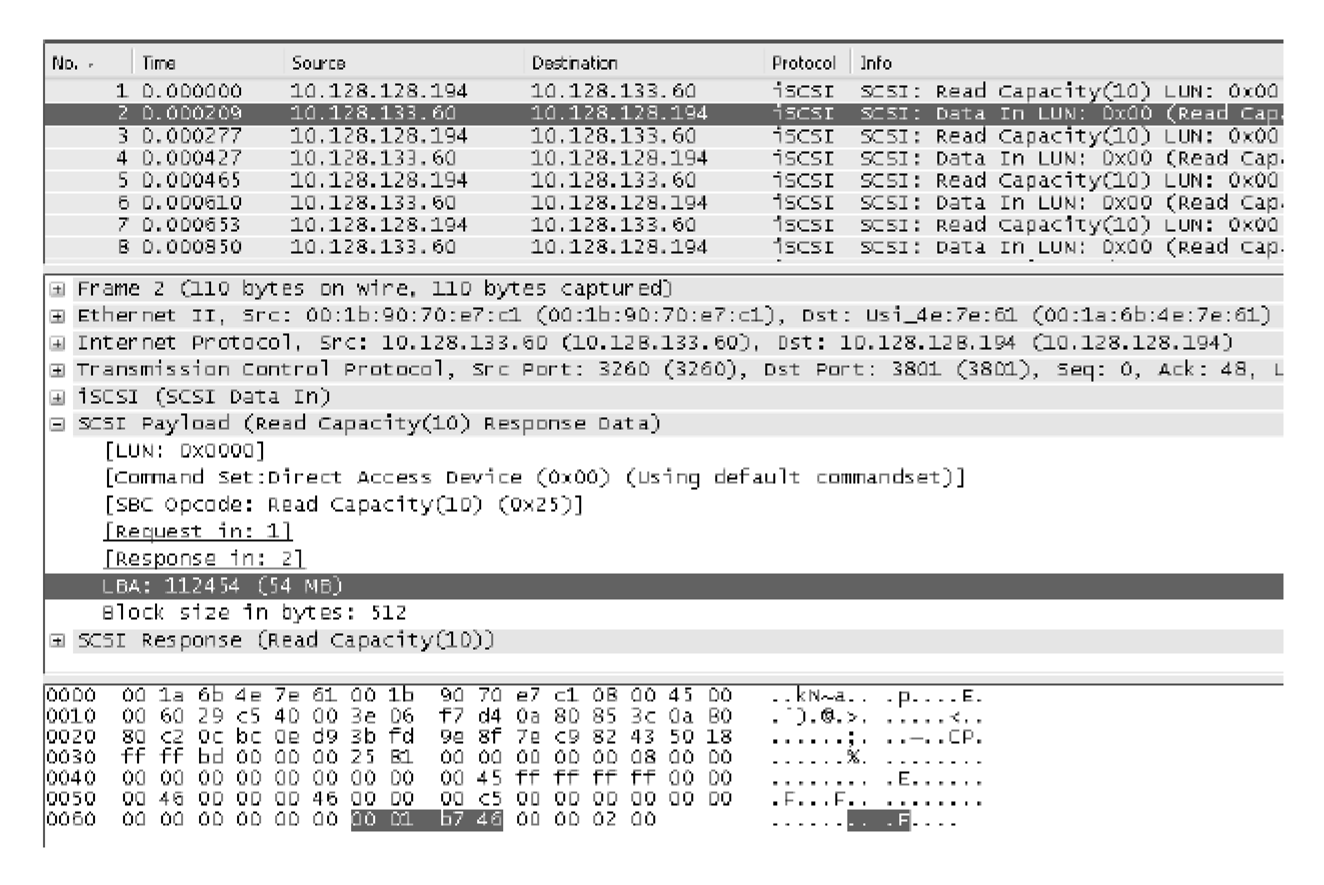


图 12.1　初始化磁盘的过程

关键帧分析

Frame1～8：ISCSI Initiator 端(主机端)首先向 LUN(ISCSI Target)发起 Read Capacity 指令来读取此 LUN 的容量信息。在 Target 返回的数据中，可以看到这个 LUN 共包含的 LBA 数为 112454 个，总容量为 54MB(112454×512÷1024÷1024)。主机连续对 Target 发出了 4 次 Read Capacity 指令，这也是程序上的设计，可能是为了充分保证读取到的容量是准确无误的。

图 12.2 包含了 Frame9～Frame10。

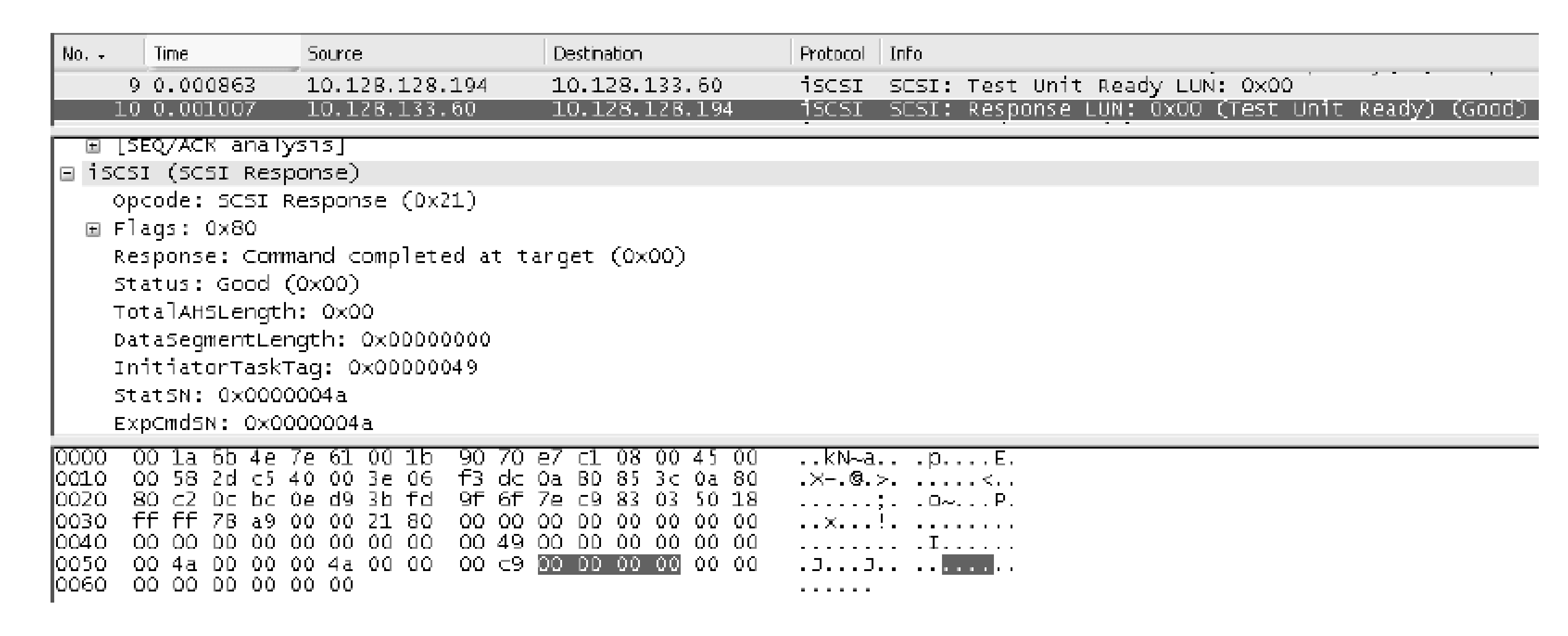


图 12.2　Test Unit Ready

Frame9～10：主机在读取完 Target 端的容量信息之后，便发起了一个 Test Unit Ready 指令来探寻 Target 端是否处于可工作状态或已经准备就绪。第 10 帧是 Target 对主机的响应。

图 12.3 包含了 Frame11～Frame13。

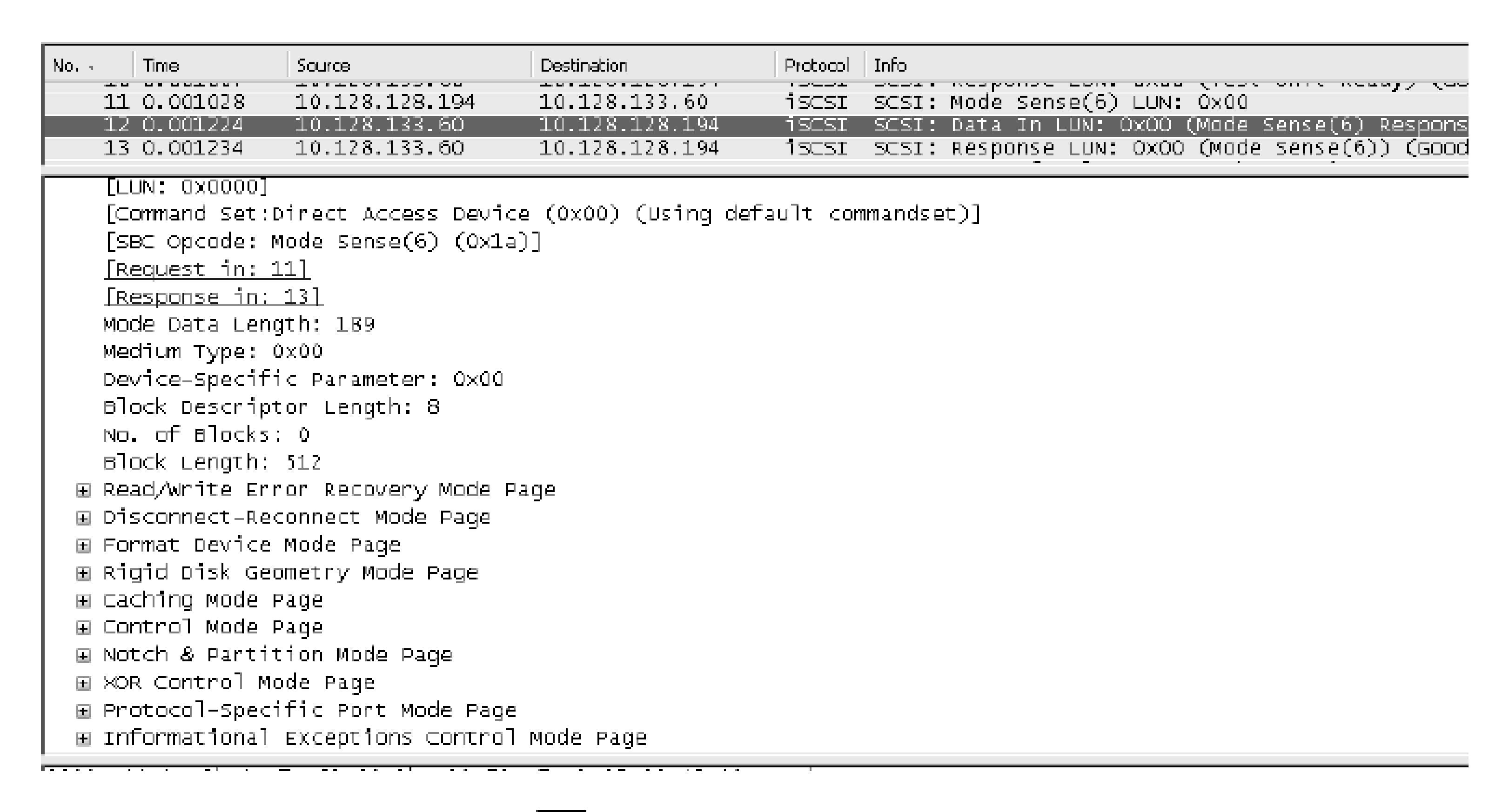


图12.3 Mode Sense

Frame11～13：主机向 Target 发出了 Mode Sense 指令，用来查询 Target 在 SCSI 处理逻辑以及物理上的相关参数。Target 在第 12 帧中返回了结果。

Frame14 为 TCP 底层的 ACK，不必深究。

图 12.4 中包含了 Frame15～Frame16。

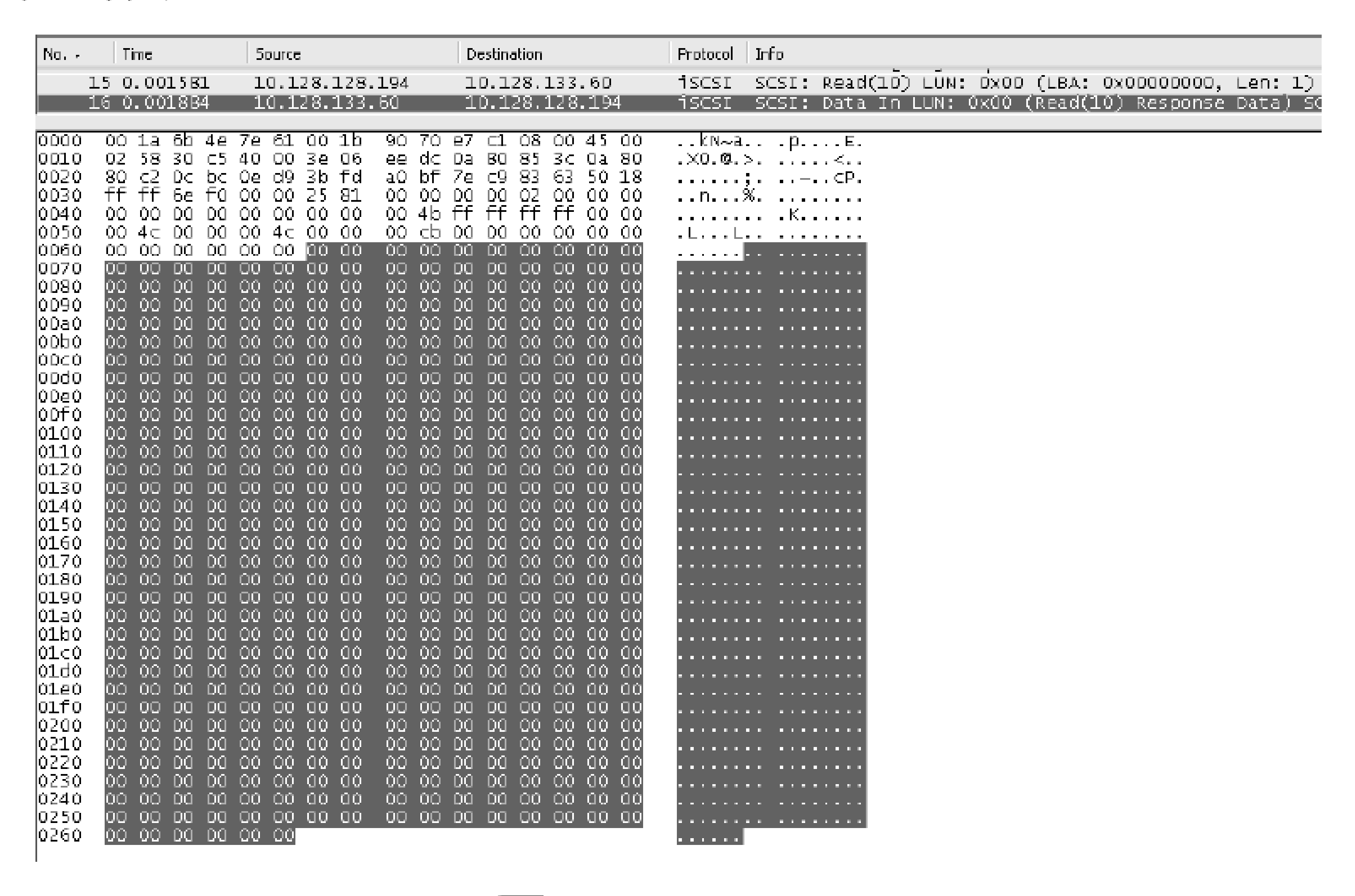


图12.4 读取 LBA0

Frame15～16：主机读取 Target 的第一个 LBA，即编号为全 0 的 LBA，这个 LBA 也就是 MBR 扇区。从数据内容中可以看出，新磁盘的这个扇区的内容是全 0。

图 12.5 中是 Frame17。

Frame17：主机向 Target 的 0 号 LBA 写入了数据，也就是将一些必要信息写入了 MBR。

图 12.6 中包含了 Frame18～Frame57。

Frame18～57：在写入 MBR 之后，主机又接连多次发出 Read LBA0、Read Capacity、Write LBA0、Test Unit Ready 指令。在最后一个 Test Unit Ready 指令发出并获得返回数据之后，磁盘初始化完毕。此时没有数据包交互了。

对磁盘分区、格式化等操作过程，这里就简单地介绍这么多。有兴趣的读者可以自行实验。

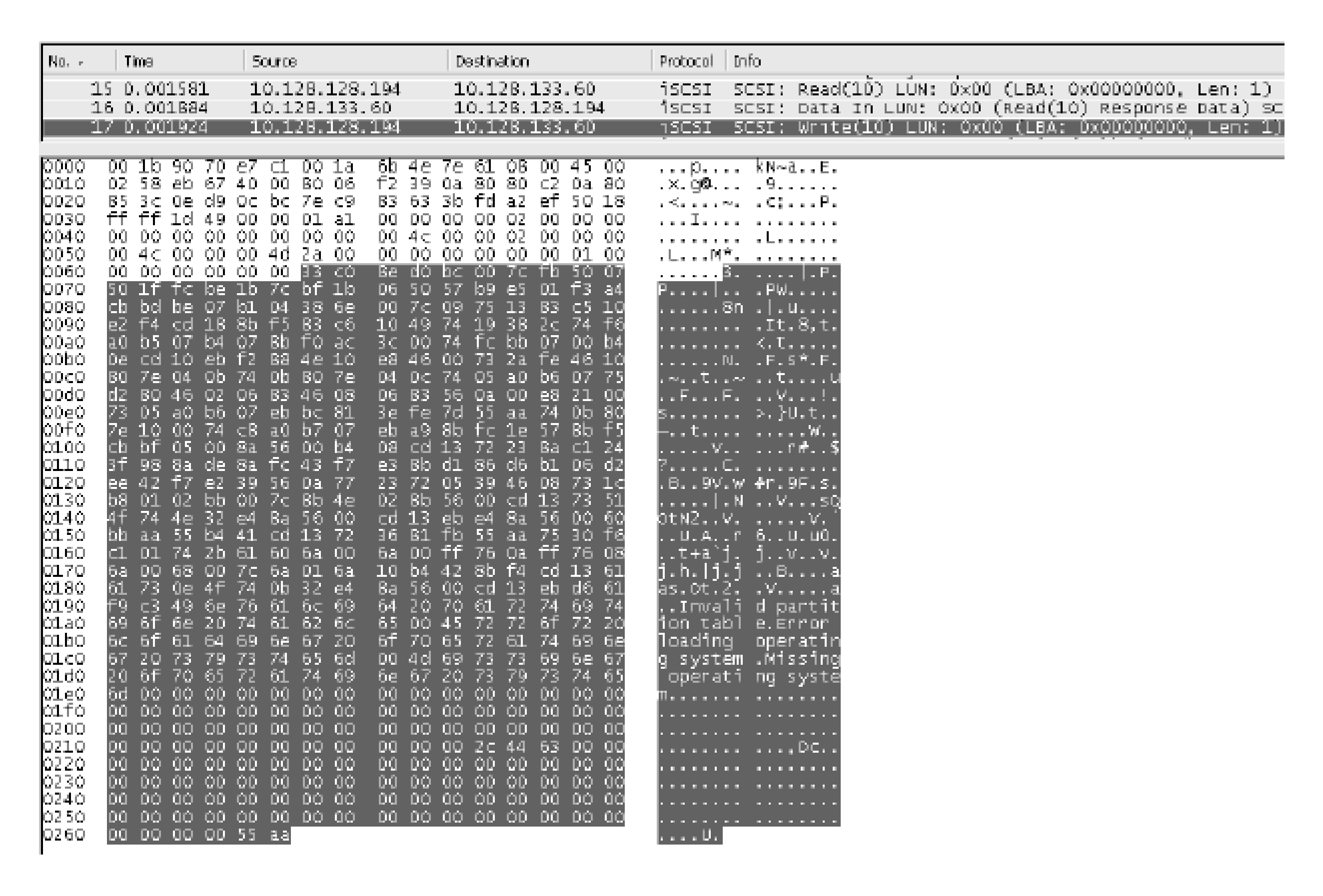


图12.5 写入 MBR

No.	Time	Source	Destination	Protocol	Info
18	0.002617	10.128.133.60	10.128.128.194	iSCSI	SCSI: Response LUN: 0x00 (Write(10)) (Good)
19	0.002670	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 1)
20	0.002943	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read(10) Response Data) SCSI: Re
21	0.002977	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 1)
22	0.003249	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read(10) Response Data) SCSI: Re
23	0.003293	10.128.128.194	10.128.133.60	iSCSI	SCSI: Write(10) LUN: 0x00 (LBA: 0x00000009, Len: 1)SCSI:
24	0.004619	10.128.133.60	10.128.128.194	iSCSI	SCSI: Response LUN: 0x00 (Write(10)) (Good)
25	0.004651	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
26	0.004796	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
27	0.004827	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 1)
28	0.005096	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read(10) Response Data) SCSI: Re
29	0.005124	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 1)
30	0.005395	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read(10) Response Data) SCSI: Re
31	0.005429	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
32	0.005584	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
33	0.005611	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read(10) LUN: 0x00 (LBA: 0x00000000, Len: 1)
34	0.005891	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read(10) Response Data) SCSI: Re
35	0.005948	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
36	0.006096	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
37	0.006113	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
38	0.006259	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
39	0.006276	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
40	0.006431	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
41	0.006446	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
42	0.006591	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
43	0.006625	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
44	0.006771	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
45	0.006789	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
46	0.006935	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
47	0.006981	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
48	0.007128	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
49	0.007159	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
50	0.007306	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
51	0.007347	10.128.128.194	10.128.133.60	iSCSI	SCSI: Read Capacity(10) LUN: 0x00
52	0.007483	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Read Capacity(10) Response Data)
53	0.007499	10.128.128.194	10.128.133.60	iSCSI	SCSI: Test Unit Ready LUN: 0x00
54	0.007644	10.128.133.60	10.128.128.194	iSCSI	SCSI: Response LUN: 0x00 (Test Unit Ready) (Good)
55	0.007668	10.128.128.194	10.128.133.60	iSCSI	SCSI: Mode Sense(6) LUN: 0x00
56	0.007875	10.128.133.60	10.128.128.194	iSCSI	SCSI: Data In LUN: 0x00 (Mode Sense(6) Response Data) SCS
57	0.150698	10.128.128.194	10.128.133.60	TCP	3801 > 3260 [ACK] Seq=2320 Ack=4968 Win=65031 Len=0

图12.6 多种指令再次读写

12.4.2 实例二：新建一个文本文档

在图 12.7 的 10.128.134.107 这台主机通过 TCP/IP 网络与 10.128.133.60 那台磁盘阵列通信。并在磁盘阵列的一个 LUN 中建立了一个新文本文件，图 12.7 所示为主机与磁盘阵列交互数据的过程。

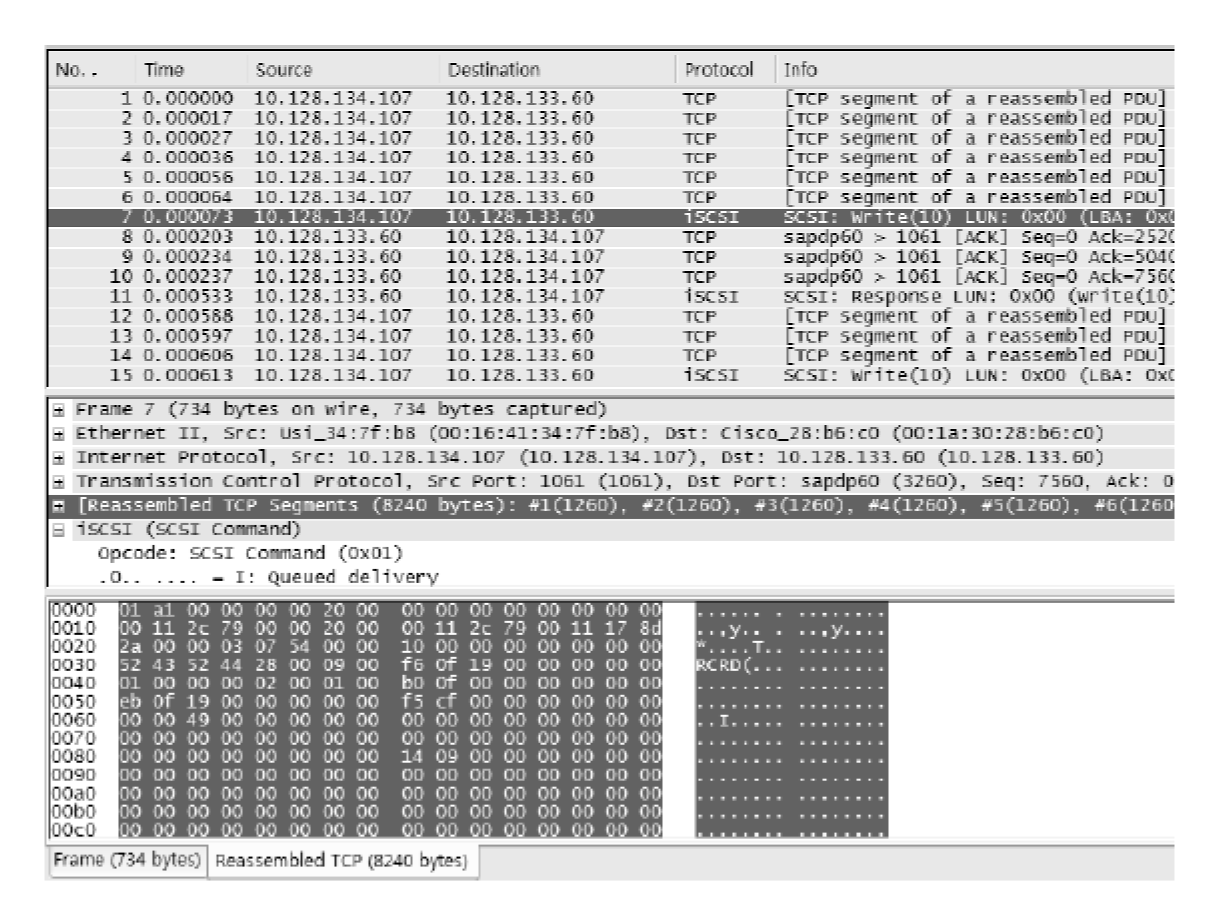


图12.7 一次写 IO

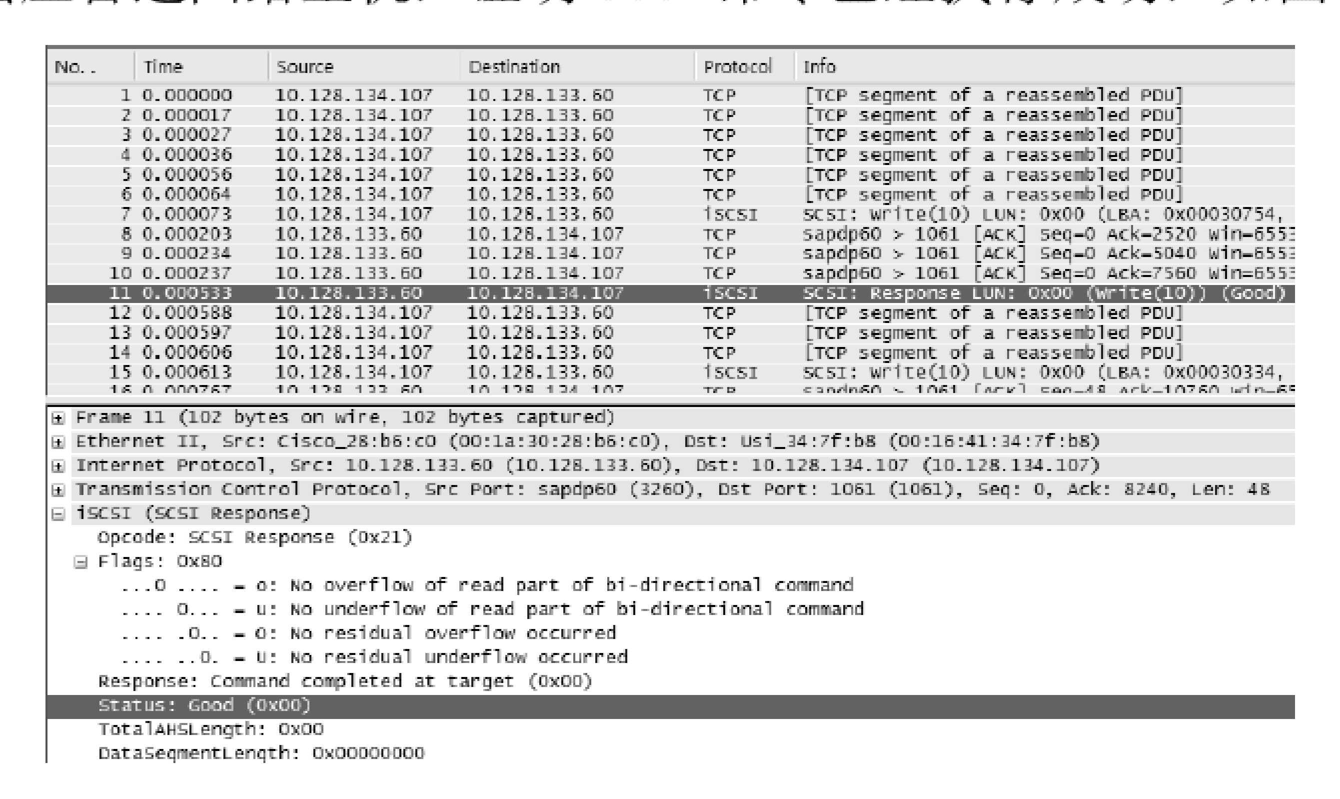
前 7 个数据帧所包含的 Payload 字节数为 8240。这 8240 个字节，就是主机向磁盘阵列所发送的一次写 IO。协议分析软件自行判断了数据帧中的协议，并定界分析了前 7 个数据帧为一次 ISCSI 的写动作。这 8240 个字节，其实就是 ISCSI 协议模块通过一次 Socket 调用 TCP/IP 协议向外发送的字节，而 TCP 根据 MSS 值，又将 8240 字节的数据分割成了 7 个数据包传送出去。这 8240 字节中包含一个 48 字节长的 ISCSI 头，以及剩余部分最终都需要被写入 LUN 的数据。我们可以计算一下，8240 字节减去 48 字节，等于 8192 字节，恰好是 16 个磁盘扇区(16×512 字节)的大小。也就是说，这次传输其实就是主机向这个 LUN 的 16 个连续扇区写入了数据。图 12.8 中也显示了相关字段的值，确实为 16。

```
  7 0.000073  10.128.134.107   10.128.133.60        iSCSI    SCSI: Write(10)
    ImmediateData: 5243524428000900F60F1900000000000010000000002000100...
    Response in: 11
⊟ SCSI CDB Write(10)
    [LUN: 0x0000]
    [Command Set:Direct Access Device (0x00) (Using default commandset)]
    [Response in: 11]
    Opcode: Write(10) (0x2a)
  ⊟ Flags: 0x00
      000. .... = WRPROTECT: 0x00
      ...0 .... = DPO: Disable page out is DISABLED (cache this data)
      .... 0... = FUA: Read from cache if possible
      .... ..0. = FUA_NV: Read from volatile or non-volatile cache permitted
    Logical Block Address (LBA): 198484
    ...0 0000 = Group: 0x00
    Transfer Length: 16
    Vendor Unique = 0, NACA = 0, Link = 0
```

图12.8 此 IO 块大小为 16 个 LBA

如图 12.8 在主机向存储设备传送的 SCSI 命令中，Opcode 字段给出了这个命令的操作代码，0x2A 表示写，如果为读，则对应的值为 0x28。同时也给出了要写入的初始 LBA 为(也就是扇区号)198484 和 Transfer Length 为 16。也就是说，主机在这条命令中通知存储设备，将随后传送的数据写入从 198484 号 LBA 开始的随后 16 个扇区中。

存储设备上的 TCP 程序返回给主机 3 个 ACK 应答开销数据包后，在第 11 帧中，将 ISCSI 协议自身的会话层应答返回给主机，证明 SCSI 命令已经执行成功，如图 12.9 所示。



No.	Time	Source	Destination	Protocol	Info
1	0.000000	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
2	0.000017	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
3	0.000027	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
4	0.000036	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
5	0.000056	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
6	0.000064	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
7	0.000073	10.128.134.107	10.128.133.60	iSCSI	SCSI: Write(10) LUN: 0x00 (LBA: 0x00030754,
8	0.000203	10.128.133.60	10.128.134.107	TCP	sapdp60 > 1061 [ACK] Seq=0 Ack=2520 Win=6553
9	0.000234	10.128.133.60	10.128.134.107	TCP	sapdp60 > 1061 [ACK] Seq=0 Ack=5040 Win=6553
10	0.000237	10.128.133.60	10.128.134.107	TCP	sapdp60 > 1061 [ACK] Seq=0 Ack=7560 Win=6553
11	0.000533	10.128.133.60	10.128.134.107	iSCSI	SCSI: Response LUN: 0x00 (Write(10)) (Good)
12	0.000588	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
13	0.000597	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
14	0.000606	10.128.134.107	10.128.133.60	TCP	[TCP segment of a reassembled PDU]
15	0.000613	10.128.134.107	10.128.133.60	iSCSI	SCSI: Write(10) LUN: 0x00 (LBA: 0x00030334,

```
⊞ Frame 11 (102 bytes on wire, 102 bytes captured)
⊞ Ethernet II, Src: Cisco_28:b6:c0 (00:1a:30:28:b6:c0), Dst: Usi_34:7f:b8 (00:16:41:34:7f:b8)
⊞ Internet Protocol, Src: 10.128.133.60 (10.128.133.60), Dst: 10.128.134.107 (10.128.134.107)
⊞ Transmission Control Protocol, Src Port: sapdp60 (3260), Dst Port: 1061 (1061), Seq: 0, Ack: 8240, Len: 48
⊟ iSCSI (SCSI Response)
    Opcode: SCSI Response (0x21)
  ⊟ Flags: 0x80
      ...0 .... = o: No overflow of read part of bi-directional command
      .... 0... = u: No underflow of read part of bi-directional command
      .... .0.. = O: No residual overflow occurred
      .... ..0. = U: No residual underflow occurred
    Response: Command completed at target (0x00)
    Status: Good (0x00)
    TotalAHSLength: 0x00
    DataSegmentLength: 0x00000000
```

图12.9 成功返回

我们来计算一下主机向存储发送了 16 个扇区的数据，其所耗费的开销大致为多少。

- 开销 1：ISCSI 头部。ISCSI 头部长 48 字节，每次 IO 都只耗费 48 字节。
- 开销 2：TCP/IP 头部以及 ACK 开销。在 MSS 值与 MTU 值适配的情况下，每个数据帧均会包含 40 字节的开销；MSS 值大于 MTU 值的情况下，IP 会将数据包拆分，第一个拆分包耗费 40 字节开销，随后的拆分包每个只耗费 20 字节开销。ACK 包的数量视窗口大小以及两端协议状态而定，数量不定。每个纯 ACK 包耗费 60 字

节开销。

- 开销 3：以太网帧头部开销。每个以太网帧耗费 14 字节开销。

根据以上描述，上述 IO 共耗费的开销为 606 字节。所以，大致估算出的开销比例为 606/(606+8192)=6%。当然这个估算是不符合统计学原理的，在这里只是大致估算一下而已。

紧接着，主机又向存储设备发出了写入请求，这次写入的则是 8 个扇区，如图 12.10 所示。

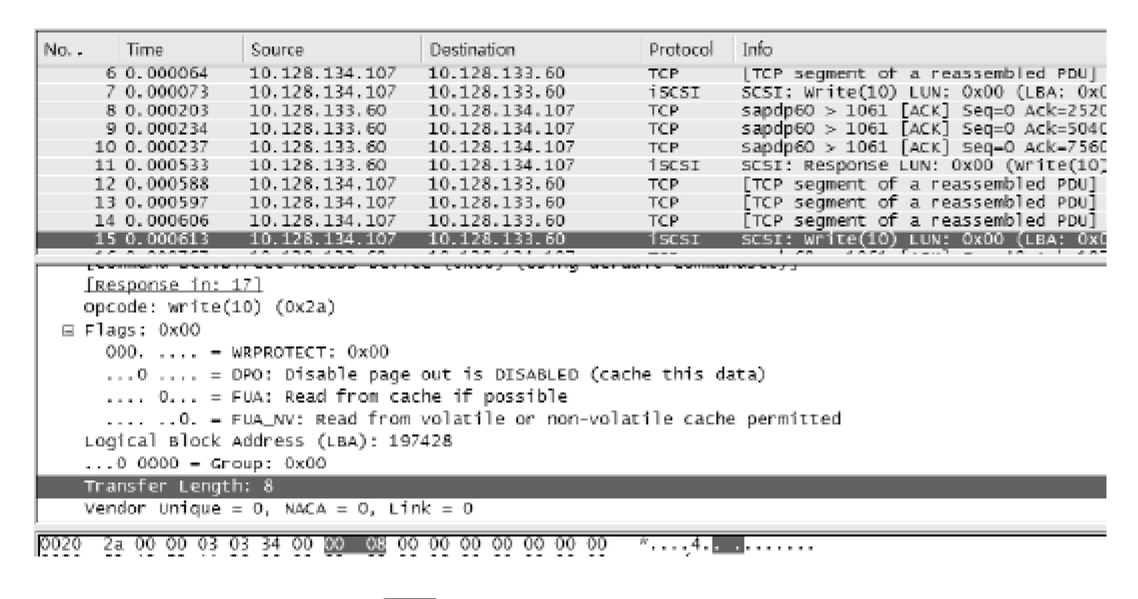


图12.10 写请求

> **猜测** 第一次主机向 LUN 写入了 16 个扇区的内容，是这个新建文本文件的实际数据，而第二次写入 8 个扇区的内容，可能是针对这个文件的元数据。当然在此只能做一个大致的猜测。若想追究到底也并不难，但是我们在此就不做过多分析了。

下图是在向某 LUN 复制一个大文件时抓取的数据包。如图 12.11 所示，主机每次 IO 请求写入的扇区数变成了 128，也就是 64KB 的数据。从数据包内容中，可以看出这个文件好像是一个视频文件，而且是一部流行的美国电视剧。

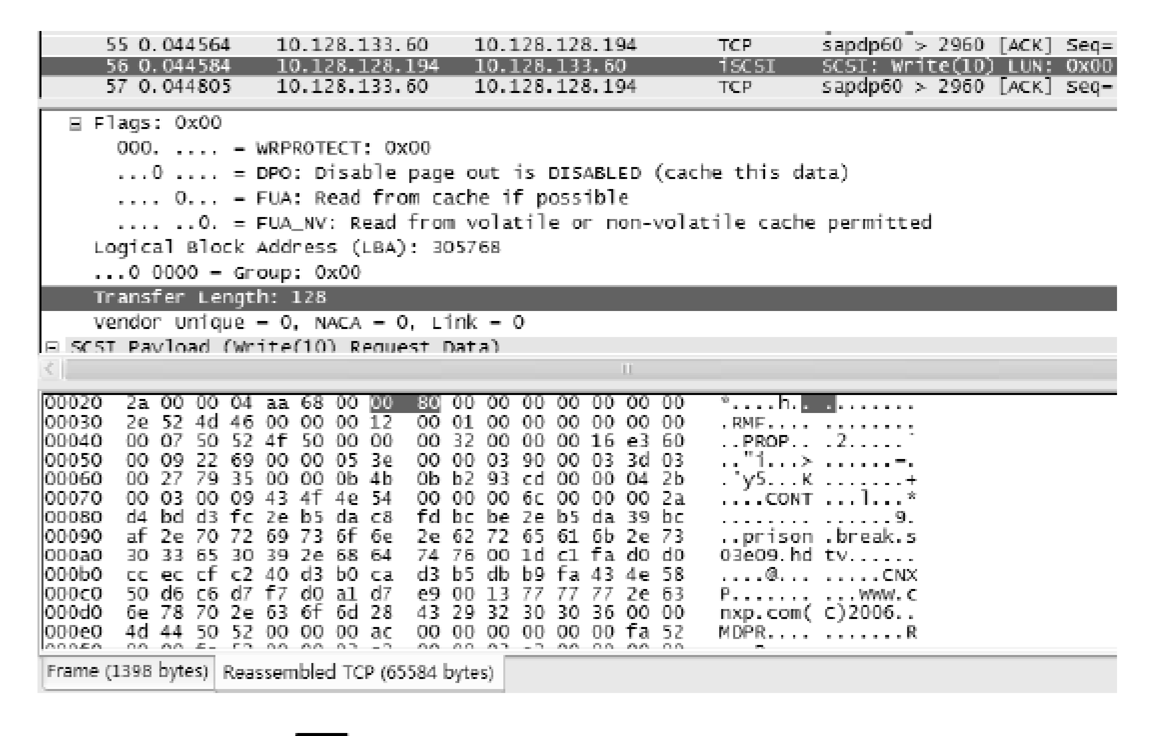


图12.11 复制一个大文件

由于 Socket 的缓冲区一般为 64KB，所以上层程序每次调用发送给 Socket 最大的数据长度也被限制在 64KB。由此就可以计算出图 12.11 中的这个 IO 将数据写入了 LBA305768～LBA305895 这 128 个扇区中。如果此时还需要接着写入新数据而同时 LBA305895 之后的扇区还是空闲状态的话，文件系统便会继续向其后的扇区继续写入。所以，图 12.12 中所示的 IO 向 LBA305896～LBA306023 这 16 个扇区写入了 64KB 的数据。如果文件依然没有

写完，则继续按照这个逻辑写下去。

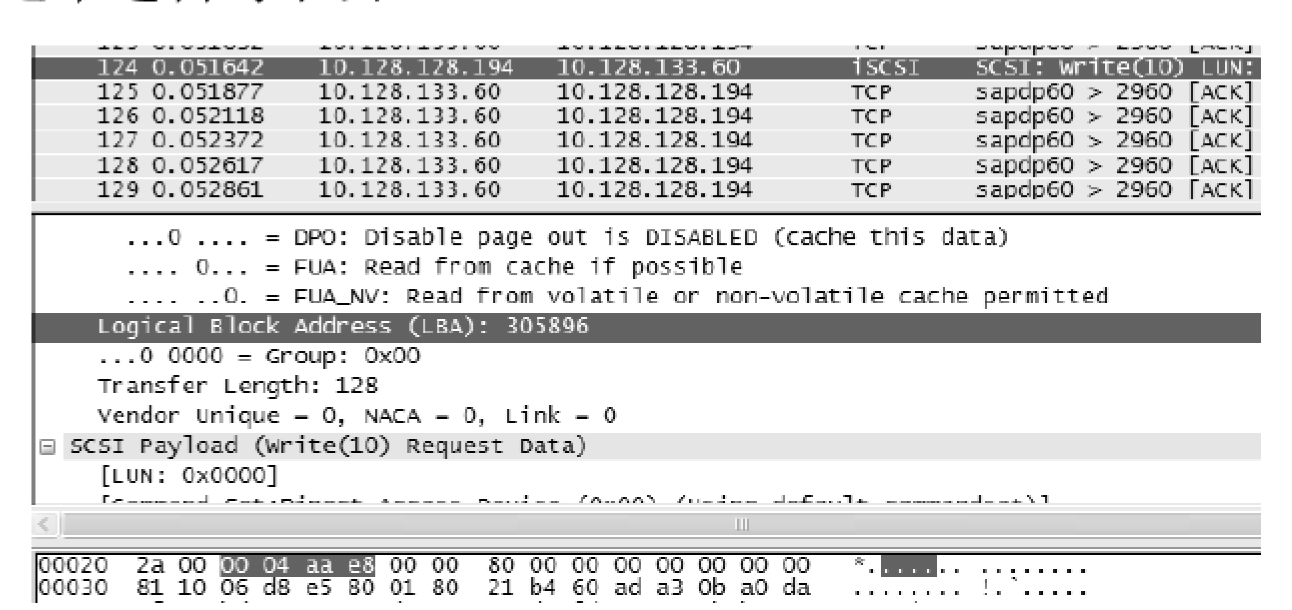


图12.12 连续的 64KB 写入

12.4.3 实例三：文件系统位图

图 12.13～图 12.16 所示的 Trace 结果是在通过主机上的文件系统，将 500MB 的数据从一个写满数据的 1GB ISCSI LUN 中删除的过程中所抓取的。

图 12.13 为删除 500MB 数据时抓取的数据包。

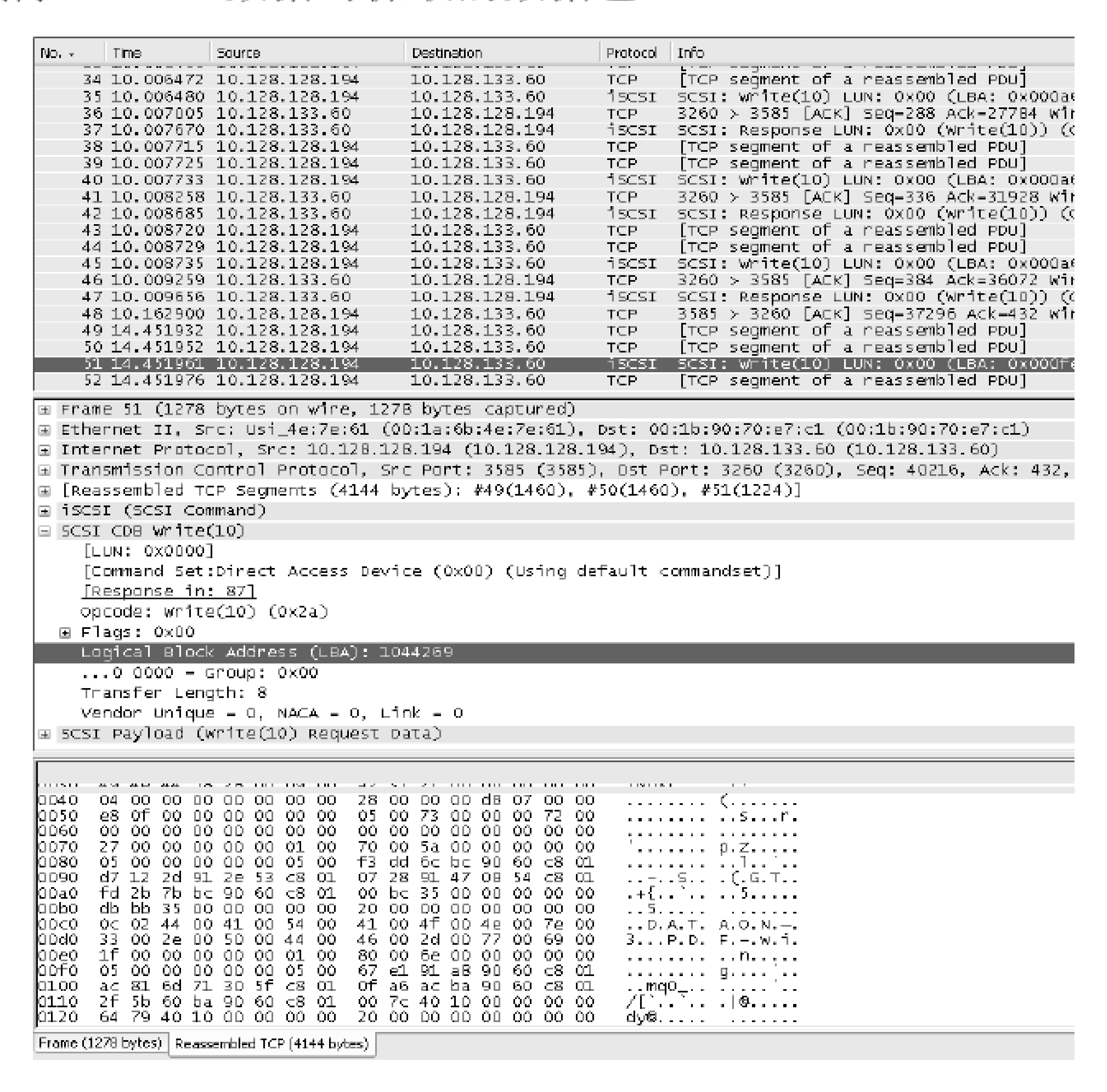


图12.13 删除 500MB 的数据

由于文件系统从磁盘上删除数据的过程中，只会修改相关的链表，从元数据中抹掉相应的记录，而不会去抹掉或者覆盖被删除的文件原来所对应的扇区上的任何数据。所以虽然删除了 500MB 的数据，但是真正的 IO 数据远小于 500MB。本例中，这个过程只交互了 163 个数据包。

在前 51 个数据包中，所有 IO 均为写操作，每个大小均为 8 个扇区(4096 字节，被分成三个数据帧)。可以判定这些 IO 其实都是在更新文件系统元数据。

从第 52～第 80 个数据包为一个 80 扇区的写 IO，如图 12.14 所示。

图12.14 文件系统位图

分析这个 IO 的实际内容，发现其包含了很多的 0xff，即二进制的 11111111。这个 IO 更新的元数据，就是本书前文中提到过的文件系统位图(见 5.7.4 节)。位图是一个元数据文件，其中用每个位来表示对应磁盘分区中的每个块(或者簇，视设计不同而定)是否正在被某个文件所占用。由于本例中，LUN 上依然留有一半的数据，所以依然有一半的簇被文件占用着。这些簇也就是图中被标明为 0xff 字节所对应的簇。被标明 0x00 的字节则对应着磁盘上未被文件占用的空闲块(簇)，但不是说这些空闲块中“没有数据”或者“数据为全 0”。对于一块崭新的磁盘，扇区中的数据可能为全 0，但是对于一块已经使用过的磁盘，扇区中会保留很多以前被删除文件的“尸体”。这些“尸体”会被随后的对这个扇区的写 IO 数据所覆盖。

图 12.15 所示的数据是在将那个 LUN 中剩余的数据全部删除的过程中所抓取的。

可以看出，LBA1044425 以及随后(包括其自身)的 56 个扇区为存放位图的扇区(当然这只是一部分，还可能有其他扇区也用来存放位图)。由于删除文件导致 LUN 上的空闲块增加，所以文件系统必须修改位图映射表，这就产生了图 12.15 中第 246 个数据包所对应的写 IO 操作。通过数据包内容可以看出，位图中几乎所有字节都为 0x00。这就说明此时 LUN 上已经几乎都是空闲块了。

读者可以自行操作并分析一下 ISCSI 以及文件系统的逻辑。但是要注意一点，由于主机上的文件系统是有缓存的，当向 LUN 做了一次操作之后，会被文件系统缓存一段时间(几秒或者十几秒都有可能)，然后才批量 Flush 到硬盘(LUN)上。所以此时一定要保持网卡抓包软件继续执行抓取，直到十几秒钟之后，流量面板中没有新的数据包被抓取到，此时方可停止抓取，获得 Trace 数据。

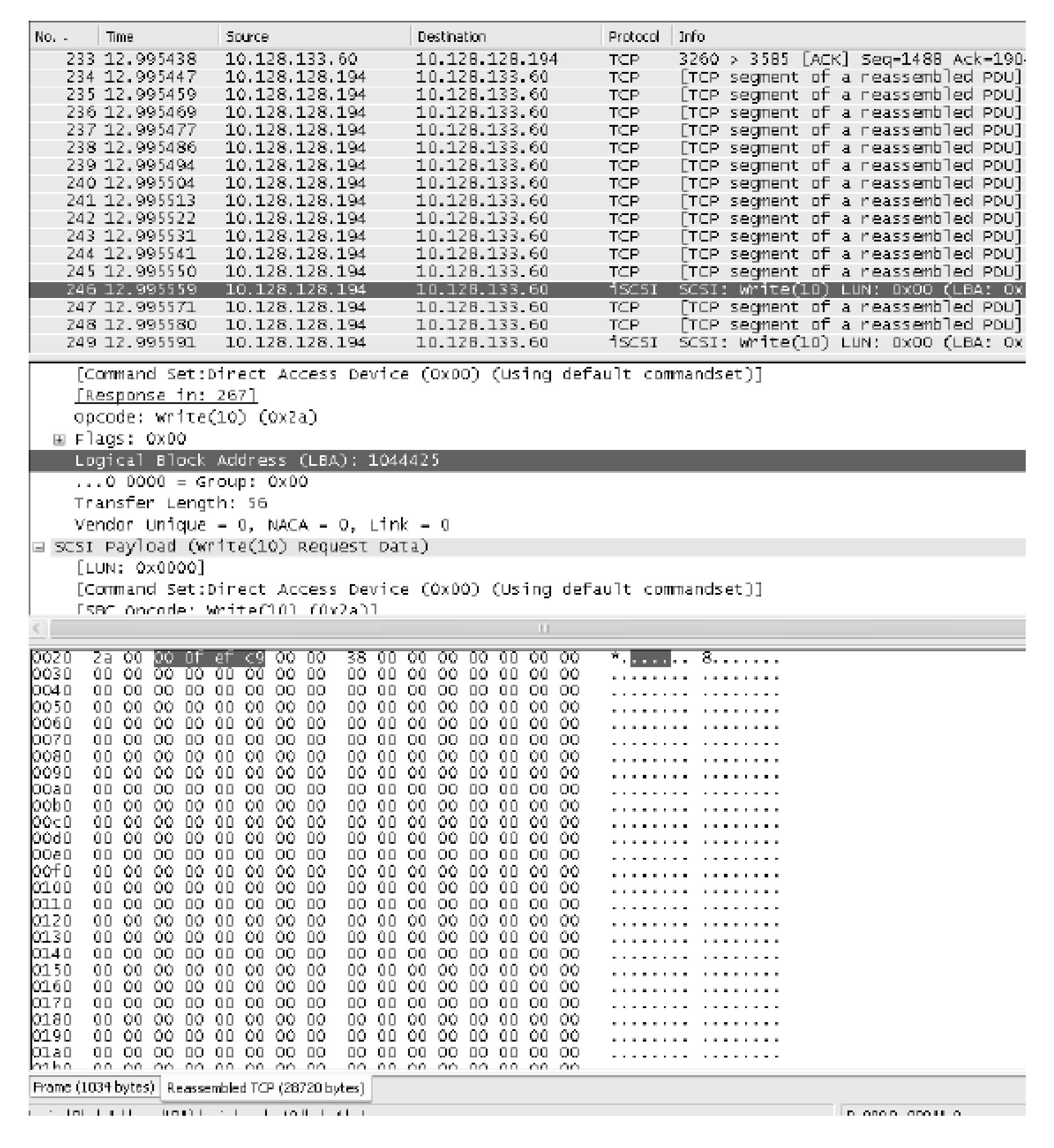


图12.15 文件系统位图

12.5 ISCSI 磁盘阵列

当年 FC 闹革命的时候，可谓是轰轰烈烈，气壮山河。当时，那场革命对磁盘阵列架构的改变真是太彻底了，所以至今人们仍然记忆犹新。而 TCP/IP 想在已经被 FC 革命过的江湖上再闹出点动静来，可就不是那么容易了。毕竟 TCP/IP 所依靠的以太网相对于 FC 来说，都属于包交换网络，而且很多概念都类似甚至相同。很多盘阵厂商并不看好 TCP/IP 的介入。但是最终还是有一批喜欢标新立异、敢于创新的厂商，举起大旗来支持 TCP/IP。

图 12.16 便是一个典型的 ISCSI 磁盘阵列的基本架构。可以看到，其前端 IO 设备就是普通的以太网卡。TCP/IP 以及 ISCSI 逻辑均运行在主内存中。后端的磁盘可以以任何方式接入总线，甚至可以是独立的磁盘阵列。磁盘经过 VM 虚拟化层之后，通过前端接口供主机访问。ISCSI 盘阵与 FC 盘阵模样差不多，只不过前端接口成了以太网口而已。NetApp 的 FAS 系列产品在同一个控制器上实现了多种协议的接口，包括 ISCSI、FCP、CIFS、NFS 以及其他的一些基于 TCP/IP 的数据访问协议。

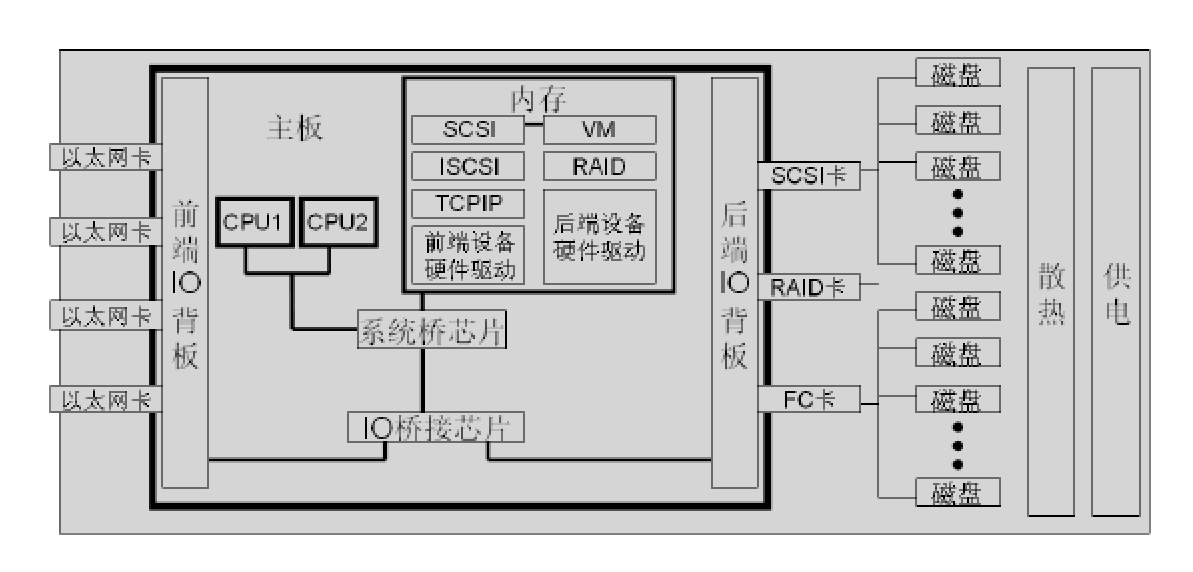


图12.16 ISCSI 盘阵架构示意图


VM 意为 Volume Manager，Virtualization Manager。

我们可以发现，这台盘阵的架构与 PC 无异。的确，不管是主机还是磁盘阵列，它们都是由计算机系统的老三样：CPU、内存、外设组成的。其所实现的功能，关键是靠所运行的软件。主机上的老三样运行的是处理业务逻辑的应用程序；而运行在盘阵老三样上的则是专门处理通过不同网络协议传输进来，或者出去的 SCSI 指令以及优化磁盘读写的程序。从本质上来说，它们二者是相同的，只不过是分工不同而已。


正因为它们本是同根生，所以相煎又何太急呢？PC 只要运行了盘阵上的软件，就是一台盘阵；相反，盘阵如果运行了应用程序，则就可以当成一台主机来使用。

目前，几乎各种操作系统都已经有了 ISCSI Initiator 软件。有些操作系统，比如 Windows、Linux 等还有了 ISCSI Target 软件。它们安装了 Target 软件，也就变成了盘阵，只不过在性能、功能和容量上没有专业盘阵强悍。

目前，TCP/IP 只是占领了盘阵前端接口的部分阵地。而对于后端磁盘接口的进攻，也不是没有设计和尝试过。当年 FC 可是一举拿下了盘阵的前端和后端。而如今老 T 能有 FC 的那个本事么？确实有些厂家的盘阵将 TCP/IP 协议作为后端磁盘到适配器之间的传输协议，但是这样的设计似乎并没有得到认可。这个结果也是可以预知的，TCP/IP 之所以可以与 FC 竞争，就是因为其优良的扩展性，而不是因为它的速度。后端需要的首先是性能，而不是扩展性。所以，后端还是乖乖地交给 FC 才是明智的选择。

12.6 IP SAN

后来，人们索性将 ISCSI 为代表的以 TCP/IP 作为传输方式的网络存储系统称作 IP SAN，即基于 IP 的存储区域网络。值得说明的是，IP SAN 并不一定要用以太网作为链路层，可以用任何支持 IP 的链路层，比如 ATM(IPoA)、PPP、HDLC，甚至是 Fibre Channel 也可以作为 IP 的链路层。

这样，就使得 IP SAN 的可扩展性变成了无限，它可以扩展到世界任何一个有 Internet 网络接入的地方。这也是 Internet Small Computer System Interface 名称的由来。

ISCSI 的方便和灵活性逐渐显现出了优势。的确，现在还有哪台主机上不带以太网适配器的？还有哪台主机上不运行 TCP/IP 协议的呢？就连大型机设备都有自己的前置 TCP/IP 处理机了。

FC 网络虽然比并行 SCSI 总线的扩展性高了很多，但是相对于 TCP/IP 的扩展性，FC 就是小巫见大巫了。

ISCSI 与 NAS 的区别

虽然 ISCSI 与 NAS 都是利用 TCP/IP+以太网来实现的。但是二者所传输的语言是大相径

庭的。NAS 传输的是文件系统语言，而 ISCSI 传输的是 SCSI 指令语言。NAS 设备上必须运行一种或者多种文件系统逻辑，才能称为 NAS；而 ISCSI Target 设备上不需要运行任何文件系统逻辑(盘阵自身操作系统文件管理除外)。

在相同的条件下，ISCSI 与 NAS 在速度与性能方面相差不大。

12.7　增强以太网和 TCP/IP 的性能

老 T 对 IP SAN 可谓是投入了一腔热血。它想尽一切办法要提高 TCP/IP 的性能，以便与 FC 抗衡。

1. Checksum Offload(CO)

计算每个 TCP 包的校验数据是一件极其枯燥乏味和耗费资源的工作。由于 TCP/IP 程序均需要运行在主机操作系统中，所以计算校验数据的任务当然要落在主机 CPU 身上。CPU 不得不拿出额外的指令周期来计算每个 TCP 包的校验数据。这对于 CPU 处理能力比较弱的主机来说，性能影响是不可忽略的。

为了将 CPU 解脱出来，一种 Checksum Offload 技术被开发出来。这种技术将计算校验数据的工作完全转移到了网卡的硬件上，对于向外发送的 TCP 包，CPU 可以不经校验直接传送给网卡，由网卡芯片来做校验；同样，对于接收到的数据包，在没提交到主机内存之前，就已经做好了校验匹配。图 12.17 就是 Checksum Offload 功能的一个图示。



图12.17　网卡的 Checksum Offload 功能

2. Large Send Offload(LSO)

TCP 需要根据底层链路的 MTU 值来适配其每次发送的数据大小。如果上层传递过来的数据大于其允许的最大分段长度(MSS 值)，则 TCP 会将这些数据分成若干个数据包发送出去，这就是所谓的 Large Send。这项工作目前也可以被转移到网卡上来完成。

3. TCP/IP Offload(TO)

TCP/IP Offload 则干脆将 TCP/IP 整个程序都放到网卡硬件芯片上来运行。这种特殊的以太网卡称为 TOE 卡，即 TCP/IP Offload Engine Card。图 12.18 是 TOE 卡在存储盘阵上的应用及架构图。

4. Security Offload(SO)

Security Offload 不仅仅将 TCP/IP 协议从主机上 Offload 了下来，它还可以在硬件上直接实现 IPSEC 相关的协议，将对数据包的加解密过程也从主机上 Offload 下来。

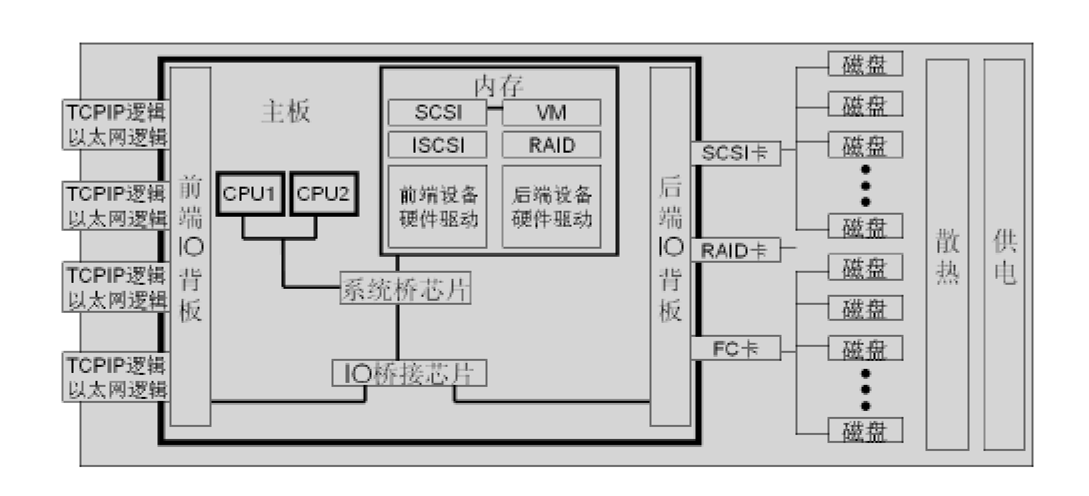


图12.18　使用 TOE 卡的 ISCSI 盘阵层次架构示意图

5. ISCSI Offload(IO)

ISCSI Offload 将 TCP/IP+ISCSI 的整套逻辑都放到网络适配卡上来运行。由于 ISCSI 的上层是 SCSI，所以一张 ISCSI 卡对于主机来说，会表现为一张 SCSI 卡。不同的是，这张虚拟的 SCSI 卡可以设置其自己独立的 IP 地址以及其他 TCP/IP 和 ISCSI 的参数。图 12.19 所示为 ISCSI 硬卡在存储阵列上的应用及架构图。

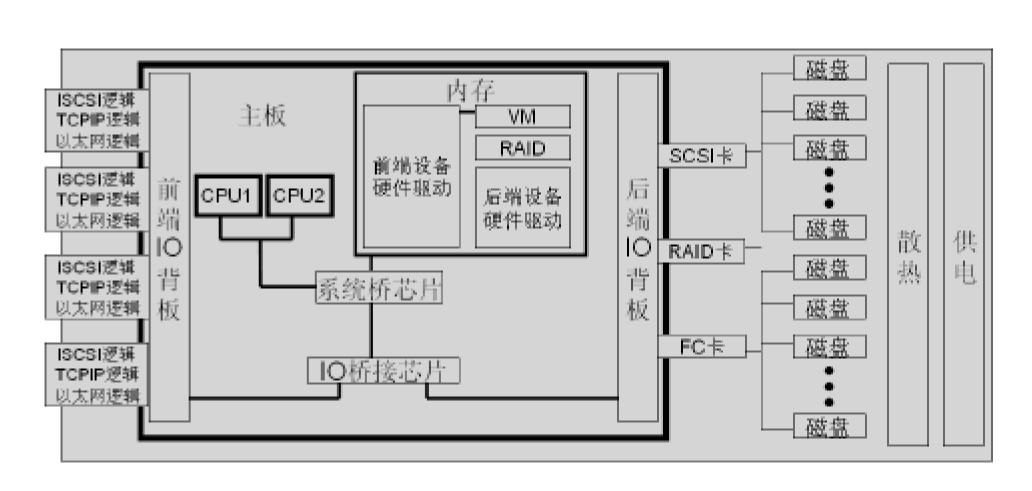


图12.19　使用 ISCSI Offload 卡的 ISCSI 盘阵架构示意图

12.8　FC SAN 节节败退

1. 成本问题

说来有点惊讶，部署 FC SAN 的成本是部署 IP SAN 的 10 倍甚至十几倍。以太网卡和以太网交换机相比 FC 卡与 FC 交换机便宜了很多。所以，对于对性能要求不是很苛刻的用户来说，部署 IP SAN 无疑是性价比最高的办法。

2. 可扩展性问题

FC 是一个专用网络。虽然 FC 当初是作为像以太网一样的通用的网络传输技术被设计出来，但是目前，其专门被用作存储网络，也算是命该如此。FC 长期被束缚在这样一个狭小的环境内，不仅造成了其不思进取的性格，而且也造成了其成本的居高不下。所以 FC 很难被扩展出去。

3. 易用性问题

曲高则和寡。既然 FC 这么不开放，那么就注定难用。部署一个 FC 存储网络比部署一个 IP 存储网络要复杂，对技术人员的要求也比较高。

4. 兼容性问题

由于 FC 极其不开放，即使 FC 有相关的标准，但不同的生产厂家出产的 FC 设备，有时

候并不一定会完全兼容，总会出现一些莫名其妙的问题。相比来说，TCP/IP 由于已经在完全开放的环境中摸爬滚打了很长的时间，所有已经被发现的 bug 也都被修复了。

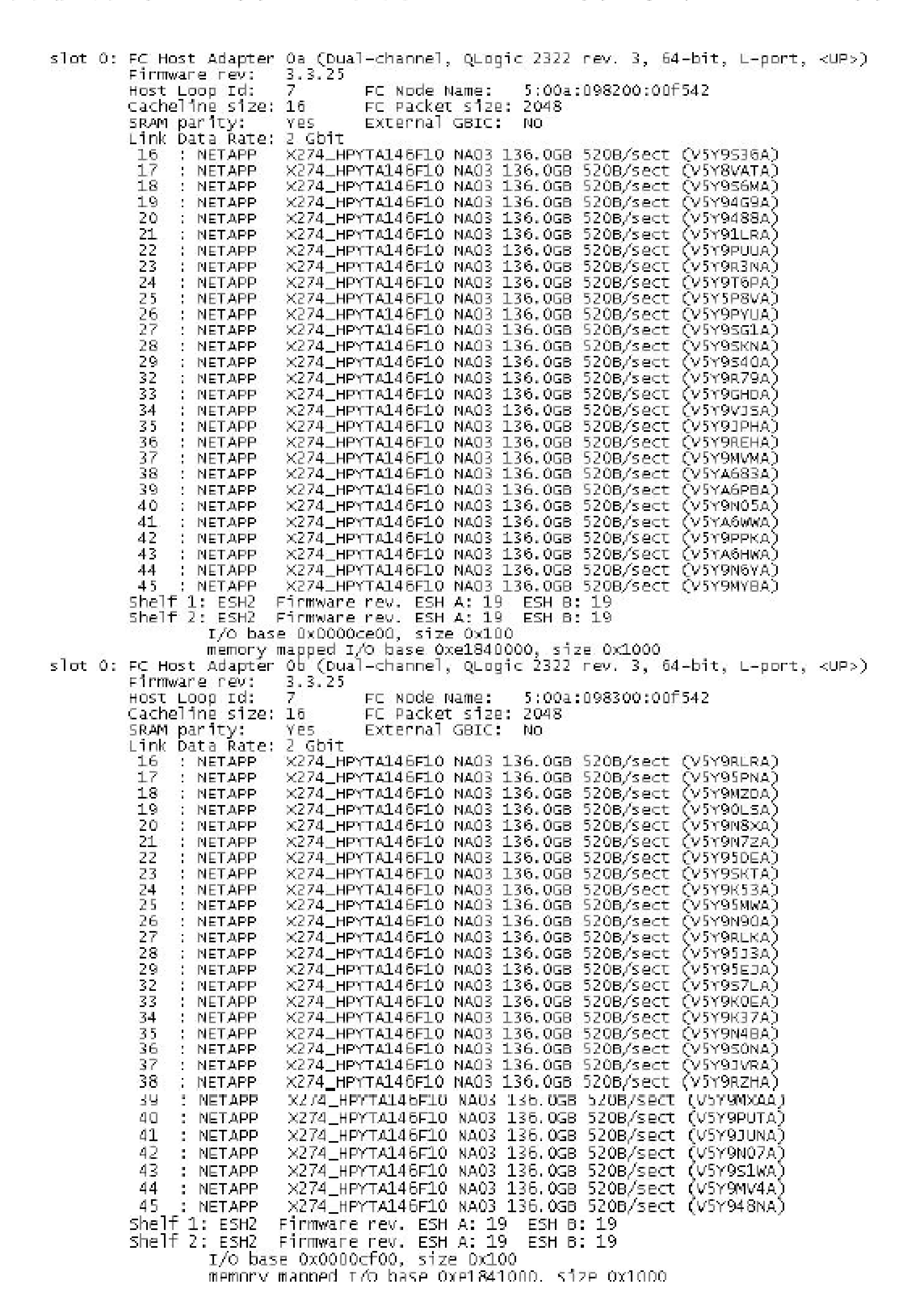
12.9 ISCSI 配置应用实例

本实例用一台 NetApp FAS3050 系列磁盘阵列充当 ISCSI 的 Target 设备，用一台运行于 Windows XP 操作系统的 PC 充当 ISCSI Initiator 端，PC 上的 ISCSI Initiator 软件为微软的 MS ISCSI Initiator2.07 版本。本例中将描述如何在存储设备上一步步的创建 LUN，然后映射给主机使用。

12.9.1 第一步：在存储设备上创建 LUN

1. 创建 Aggregate

所谓 Aggregate 是指 RaidGroup 的组合，一个 Aggr(Aggregate)可以包含多个 RG(Raid Group)。如图 12.20 所示，这台盘阵在其后端的 0a 和 0b 个 FC 通道下各连接了两台扩展柜(Shelf1 和 Shelf2)，每台扩展柜包含 14 块硬盘，这样每个通道包含了 28 块硬盘。



```
slot 0: FC Host Adapter 0a (Dual-channel, QLogic 2322 rev. 3, 64-bit, L-port, <UP>)
        Firmware rev:   3.3.25
        Host Loop Id:   7       FC Node Name:   5:00a:098200:00f542
        Cacheline size: 16      FC Packet size: 2048
        SRAM parity:    Yes     External GBIC:  No
        Link Data Rate: 2 Gbit
         16  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9S36A)
         17  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y8VATA)
         18  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9S6MA)
         19  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y94G9A)
         20  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9488A)
         21  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y91LRA)
         22  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9PUUA)
         23  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9R3NA)
         24  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9T6PA)
         25  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y5P8VA)
         26  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9PYUA)
         27  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9SG1A)
         28  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9SKNA)
         29  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9S40A)
         32  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9R79A)
         33  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9GHDA)
         34  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9VJSA)
         35  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9JPHA)
         36  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9REHA)
         37  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9MVMA)
         38  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5YA683A)
         39  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5YA6PBA)
         40  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N05A)
         41  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5YA6WWA)
         42  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9PPKA)
         43  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5YA6HWA)
         44  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N6YA)
         45  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9MYBA)
        Shelf 1: ESH2  Firmware rev. ESH A: 19  ESH B: 19
        Shelf 2: ESH2  Firmware rev. ESH A: 19  ESH B: 19
                I/O base 0x0000ce00, size 0x100
                memory mapped I/O base 0xe1840000, size 0x1000
slot 0: FC Host Adapter 0b (Dual-channel, QLogic 2322 rev. 3, 64-bit, L-port, <UP>)
        Firmware rev:   3.3.25
        Host Loop Id:   7       FC Node Name:   5:00a:098300:00f542
        Cacheline size: 16      FC Packet size: 2048
        SRAM parity:    Yes     External GBIC:  No
        Link Data Rate: 2 Gbit
         16  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9RLRA)
         17  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y95PNA)
         18  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9MZDA)
         19  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y90LSA)
         20  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N8XA)
         21  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N7ZA)
         22  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y95DEA)
         23  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9SKTA)
         24  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9K53A)
         25  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y95MWA)
         26  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N90A)
         27  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9RLKA)
         28  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y95J3A)
         29  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y95EJA)
         32  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9S7LA)
         33  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9K0EA)
         34  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9K37A)
         35  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N4BA)
         36  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9SONA)
         37  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9JVRA)
         38  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9RZHA)
         39  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9MXAA)
         40  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9PUTA)
         41  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9JUNA)
         42  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9N07A)
         43  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9S1WA)
         44  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y9MV4A)
         45  : NETAPP   X274_HPYTA146F10 NA03 136.0GB 520B/sect (V5Y948NA)
        Shelf 1: ESH2  Firmware rev. ESH A: 19  ESH B: 19
        Shelf 2: ESH2  Firmware rev. ESH A: 19  ESH B: 19
                I/O base 0x0000cf00, size 0x100
                memory mapped I/O base 0xe1841000, size 0x1000
```

图 12.20 系统硬盘列表

图 12.21 所示的是系统中目前还没有被分配到 RG 中的磁盘，标为 Spare，共 16 块。

如图 12.22 所示，使用“aggr create”命令创建一个 aggr。“-r 6”参数表示每个 RG

最大允许包含 6 块磁盘，如果超过 6 块，则自动形成另一个 RG。“-t raid_dp”参数表示 RG 的 RAID 类型为 RAIDDP(这种 RAID 类型本书不做描述，具体可以在 Internet 搜索)。12 表示这个 aggr 包含 12 块盘。计算一下这样正好可以形成两个 RG，每个 RG6 块盘。“aggrtest”为这个 aggr 的名字。

```
Spare disks

RAID Disk       Device  HA   SHELF BAY CHAN Pool Type  RPM  Used (MB/blks)    Phys (MB/blks)
---------       ------  ------------- ---- ---- ---- ----- --------------    --------------
Spare disks for block or zoned checksum traditional volumes or aggregates
spare           0a.22   0a    1   6   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.23   0a    1   7   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.24   0a    1   8   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.25   0a    1   9   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.26   0a    1   10  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.27   0a    1   11  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.28   0a    1   12  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.29   0a    1   13  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.38   0a    2   6   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.39   0a    2   7   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.40   0a    2   8   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.41   0a    2   9   FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.42   0a    2   10  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.43   0a    2   11  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.44   0a    2   12  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
spare           0a.45   0a    2   13  FC:A   -  FCAL 10000 136000/278528000  137422/281442144 (not zeroed)
```

图12.21 Spare 硬盘列表

```
MelonHead> aggr create aggrtest -r 6 -t raid_dp 12
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.43 Shelf 2 Bay 11 [NETAP
P   X274_HPYTA146F10 NA03] S/N [V5YA6HWA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.27 Shelf 1 Bay 11 [NETAP
P   X274_HPYTA146F10 NA03] S/N [V5Y9SG1A] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.42 Shelf 2 Bay 10 [NETAP
P   X274_HPYTA146F10 NA03] S/N [V5Y9PPKA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.26 Shelf 1 Bay 10 [NETAP
P   X274_HPYTA146F10 NA03] S/N [V5Y9PYUA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.41 Shelf 2 Bay 9 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5YA6WWA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg1/0a.25 Shelf 1 Bay 9 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5Y5P8VA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.24 Shelf 1 Bay 8 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5Y9T6PA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.40 Shelf 2 Bay 8 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5Y9N05A] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.23 Shelf 1 Bay 7 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5Y9R3NA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.39 Shelf 2 Bay 7 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5YA6PBA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.22 Shelf 1 Bay 6 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5Y9PUUA] to aggregate aggrtest has completed successfully
Thu Jun 19 14:12:08 CST [MelonHead: raid.vol.disk.add.done:notice]: Addition of Disk /aggrtest/plex0/rg0/0a.38 Shelf 2 Bay 6 [NETAPP
   X274_HPYTA146F10 NA03] S/N [V5YA683A] to aggregate aggrtest has completed successfully
Creation of an aggregate with 12 disks has completed.
MelonHead> Thu Jun 19 14:12:09 CST [MelonHead: wafl.vol.add:notice]: Aggregate aggrtest has been added to the system.

MelonHead>
```

图12.22 创建 aggr

如图 12.23 所示，创建好的 aggr 中有两个分别包含 6 块盘的 RG。

```
MelonHead> sysconfig -r
Aggregate aggrtest (online, raid_dp) (block checksums)
  Plex /aggrtest/plex0 (online, normal, active)
    RAID group /aggrtest/plex0/rg0 (normal)

      RAID Disk Device  HA  SHELF BAY CHAN Pool Type  RPM  Used (MB/blks)    Phys (MB/blks)
      --------- ------  ------------- ---- ---- ---- ----- --------------    --------------
      dparity   0a.38   0a    2   6   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      parity    0a.22   0a    1   6   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.39   0a    2   7   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.23   0a    1   7   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.40   0a    2   8   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.24   0a    1   8   FC:A   -  FCAL 10000 136000/278528000  137422/281442144

    RAID group /aggrtest/plex0/rg1 (normal)

      RAID Disk Device  HA  SHELF BAY CHAN Pool Type  RPM  Used (MB/blks)    Phys (MB/blks)
      --------- ------  ------------- ---- ---- ---- ----- --------------    --------------
      dparity   0a.25   0a    1   9   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      parity    0a.41   0a    2   9   FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.26   0a    1   10  FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.42   0a    2   10  FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.27   0a    1   11  FC:A   -  FCAL 10000 136000/278528000  137422/281442144
      data      0a.43   0a    2   11  FC:A   -  FCAL 10000 136000/278528000  137422/281442144
```

图12.23 aggrtest 中包含两个 RG

如图 12.24 所示，刚创建好的 aggr 名为 aggrtest，大小为 955GB，其中给快照预留的空间为 47GB，实际可用在线存储用户数据的空间为 908GB。

```
MelonHead> df -Ah
Aggregate                total    used    avail capacity
aggr2                    340GB    12MB    340GB      0%
aggr2/.snapshot           17GB    63MB     17GB      0%
aggr1                    340GB    20GB    320GB      6%
aggr1/.snapshot           17GB    63MB     17GB      0%
vol0                      95GB   822MB     94GB      1%
vol0/.snapshot            23GB    57MB     23GB      0%
aggrtest                 908GB   164KB    908GB      0%
aggrtest/.snapshot        47GB     0GB     47GB      0%
MelonHead>
```

图12.24 aggr 的空间分布

2. 在 aggr 中创建 Vol

Vol，即卷，是凌驾于 aggr 之上的一层虚拟化产物，目的是为了更加灵活地取用 aggr

所提供的存储空间。Vol 可以任意创建删除，任意增加或者减小容量。

如图 12.25 所示，在 aggrtest 中创建了一个名为 voltest，然后将 offline 删除。这个过程非常简单，命令发出之后立即生效。

```
MelonHead> vol create voltest aggrtest 500G
Creation of volume 'voltest' with size 500g on containing aggregate
'aggrtest' has completed.
MelonHead> vol offline voltest
Thu Jun 19 14:17:09 CST [MelonHead: wafl.vvol.offline:info]: Volume 'voltest' has been set temporarily offline
Volume 'voltest' is now offline.
MelonHead> vol destroy voltest
Are you sure you want to destroy this volume? y
Thu Jun 19 14:17:16 CST [MelonHead: wafl.vvol.destroyed:info]: Volume voltest destroyed.
Volume 'voltest' destroyed.
```

图 12.25　创建/删除一个 Vol

如图 12.26 所示，在 aggrtest 上创建两个卷：iscsi1 和 iscsi2，大小均为 300GB。

```
MelonHead> vol create iscsi1 aggrtest 300G
Creation of volume 'iscsi1' with size 300g on containing aggregate
'aggrtest' has completed.
MelonHead> vol create iscsi2 aggrtest 300G
Creation of volume 'iscsi2' with size 300g on containing aggregate
'aggrtest' has completed.
MelonHead> █
```

图 12.26　创建两个新卷

3. 在 Vol 中创建 LUN

LUN 是最终提交给主机使用的一块存储空间，NetApp 将 LUN 容纳于 Vol 的空间之下，并且 LUN 也可以任意增、删、扩、缩。

如图 12.27 所示，在每个 Vol 里分别创建一个大小为 200GB 的 LUN。

```
MelonHead> lun create -s 200G -t windows /vol/iscsi1/iscsilun1
lun create: created a LUN of size:  200.0g (214778511360)
MelonHead> lun create -s 200G -t windows /vol/iscsi2/iscsilun2
lun create: created a LUN of size:  200.0g (214778511360)
MelonHead> █
```

图 12.27　创建 LUN

4. 创建 Igroup 并映射 LUN

所谓 Igroup，是用来管理 LUN－主机映射关系的。本书前文中描述过这种灵活的映射关系。Igroup 就像一个桥梁，主机和 LUN 如果都映射到某个 Igoup，那么这台主机就可以访问这些 LUN。

如图 12.28 所示，创建一个名为 IG 的 ISCSI 类型的 Igroup，并给它映射到了一个主机端的 ISCSI Initiator 的 IQN 地址。接着，将两个 LUN 都映射到了这个 Igroup 中。

```
MelonHead> igroup create -i -t windows IG iqn.1991-05.com.microsoft:dongz-xp.ab.iscsi1.com
MelonHead> lun map /vol/iscsi1/iscsilun1 IG
Thu Jun 19 16:58:00 CST [MelonHead: lun.map:info]: LUN /vol/iscsi1/iscsilun1 was mapped to initiator group IG=0
MelonHead> lun map /vol/iscsi2/iscsilun2 IG
Thu Jun 19 16:58:14 CST [MelonHead: lun.map:info]: LUN /vol/iscsi2/iscsilun2 was mapped to initiator group IG=1
MelonHead> █
```

图 12.28　创建 Igroup 并映射 LUN 和主机

12.9.2　第二步：在主机端挂载 LUN

1. 确认主机端的 IQN 名称

确认主机端 ISCSI Initiator 的 IQN 名称是否与 Igroup 中缩配置的一致，如图 12.29 所示。

2. 添加 ISCSI Target 端地址

添加 ISCSI Target 端地址，发现其上的 LUN，如图 12.30 和图 12.31 所示。图 12.32 显

示已经发现 ISCSI 目标，但处于未激活状态。

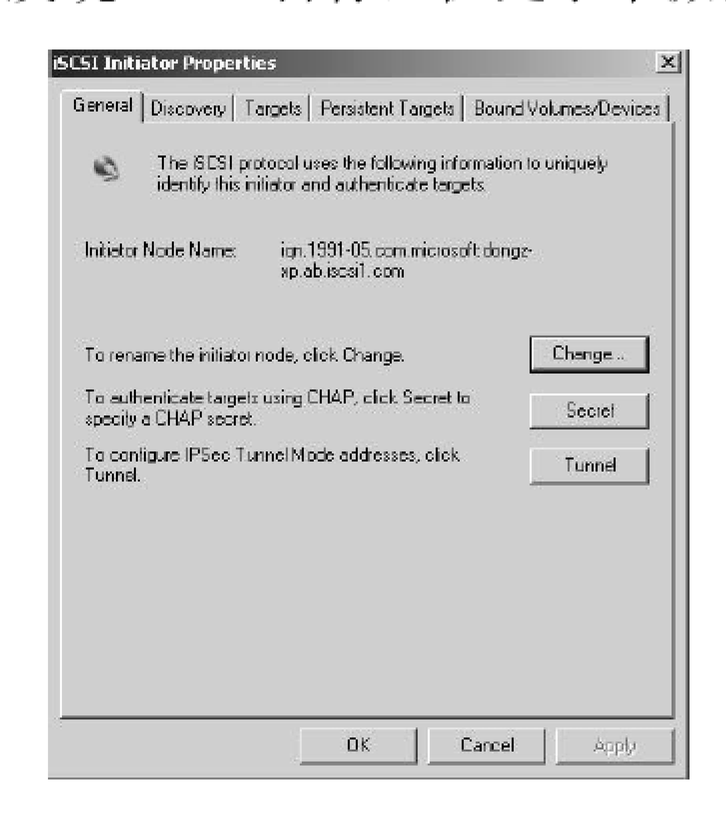


图12.29　IQN 名称

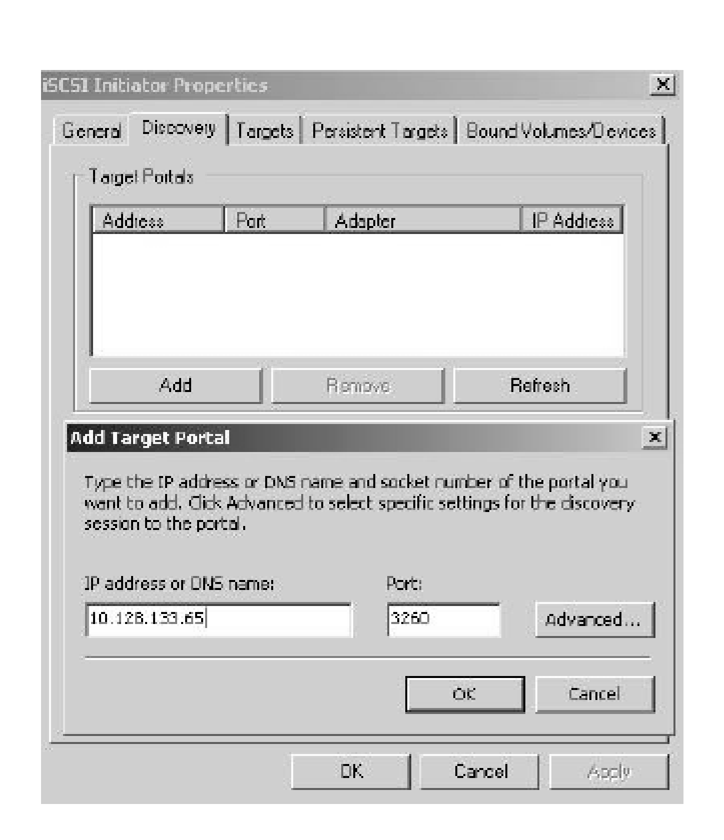


图12.30　添加 ISCSI Target 端地址

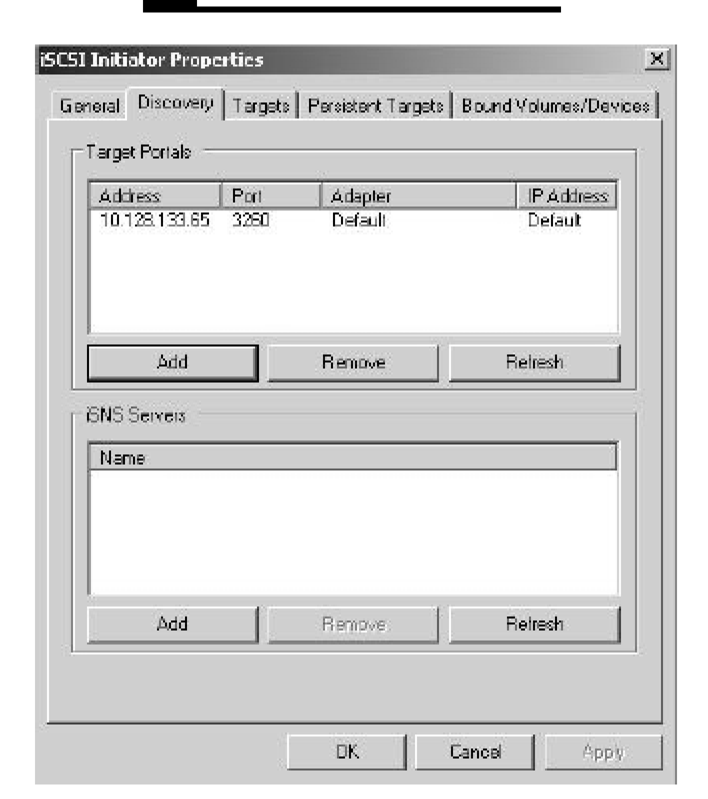


图12.31　添加完成

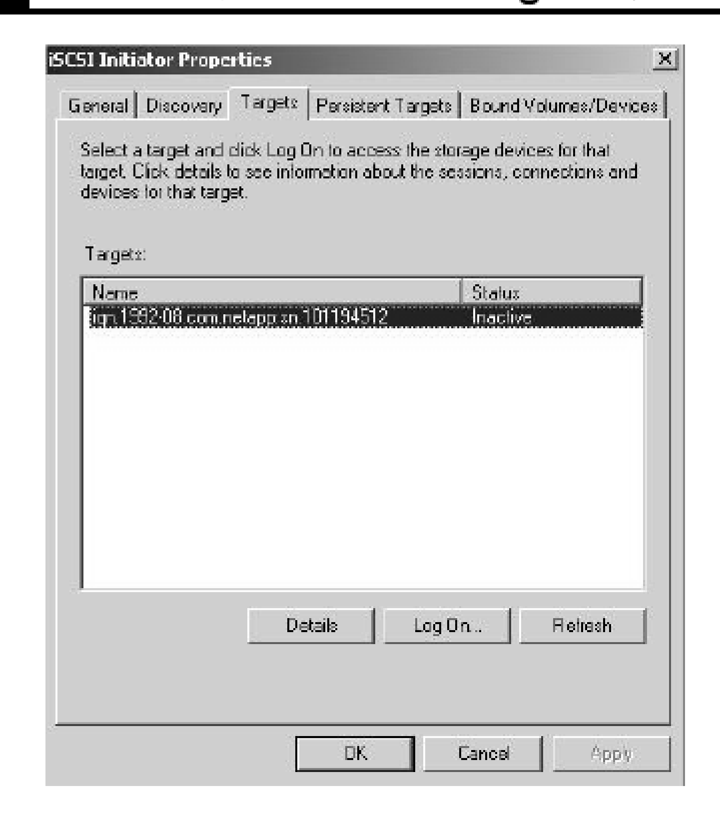


图12.32　未激活状态的 Target

3．激活 ISCSI 目标端，发现 LUN

单击图 12.32 中的 LogOn 按钮，如图 12.33 所示。

单击 OK 按钮后，可以在图 12.34 中看到，目标已经连接，此时 LUN 会被主机识别到。

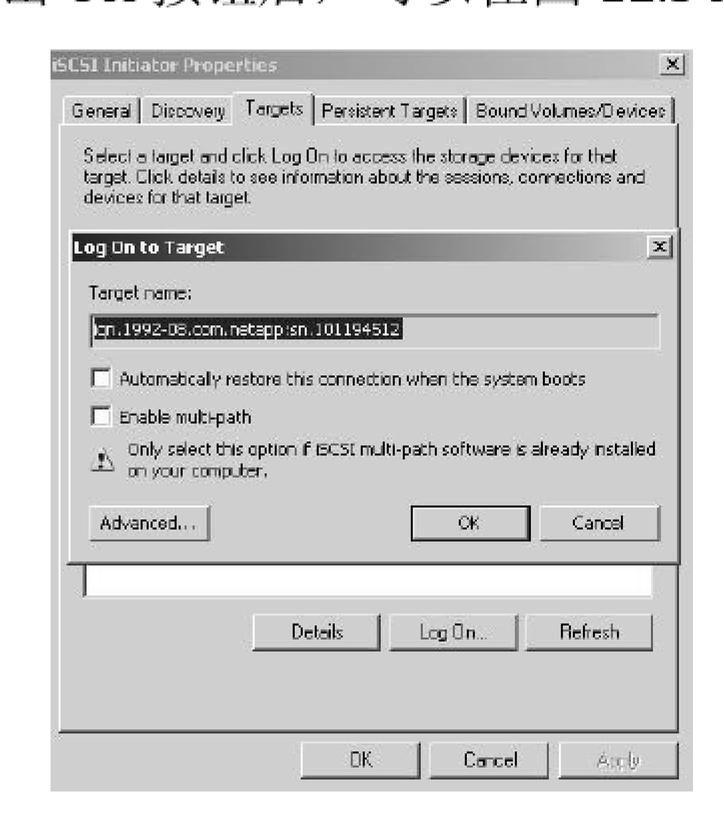


图12.33　激活 ISCSI 目标

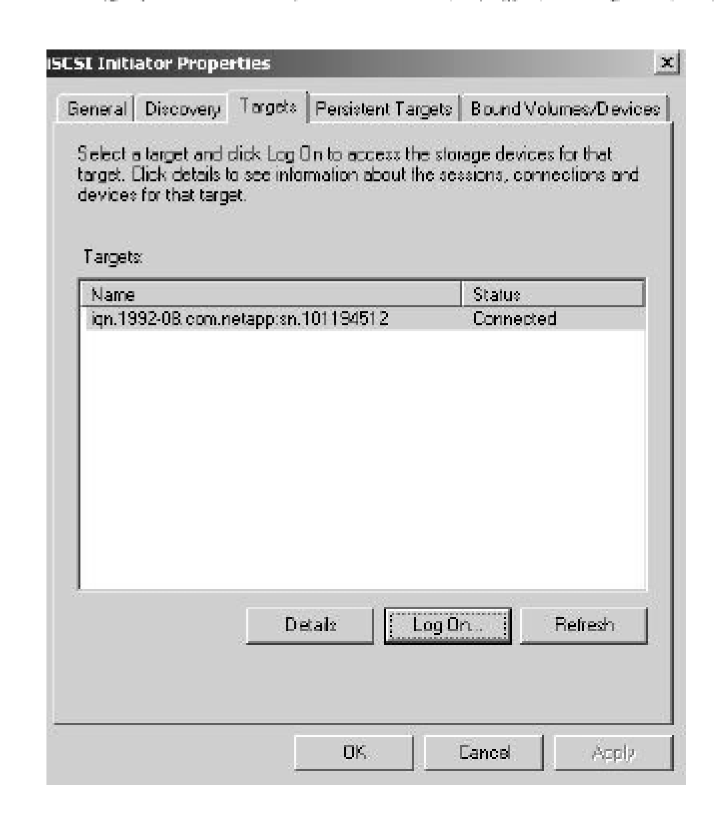


图12.34　处于激活状态的 ISCSI 目标

4．使用 LUN

如图 12.35 所示，在主机的磁盘管理器中会发现两块未初始化的磁盘，其对应的是存

储设备上的两个 LUN。

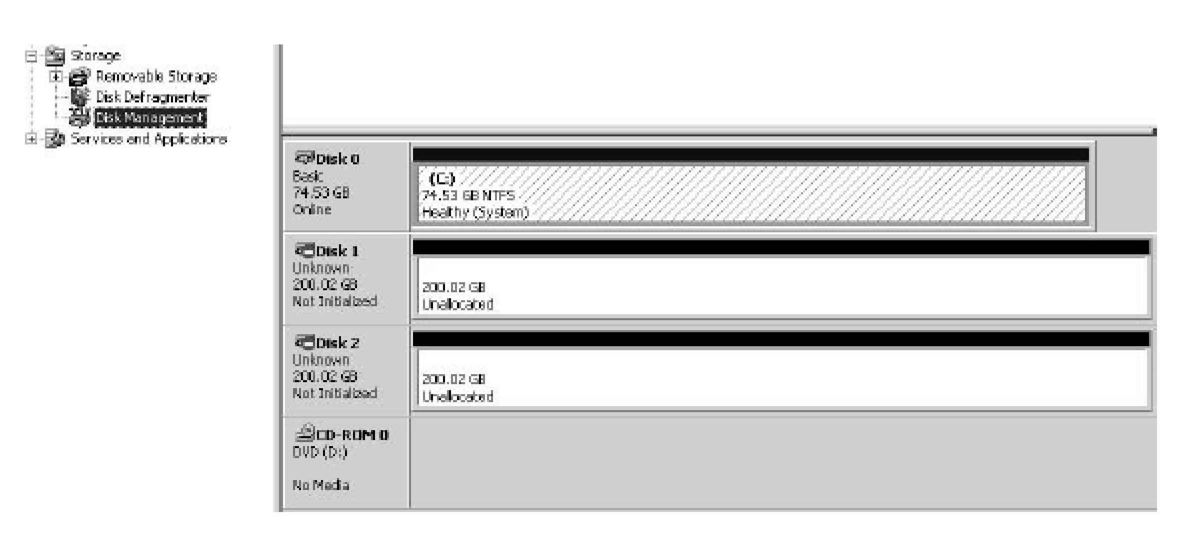


图12.35　新识别到的两个 LUN

如图 12.36 所示，初始化之后格式化这两块磁盘(LUN)。

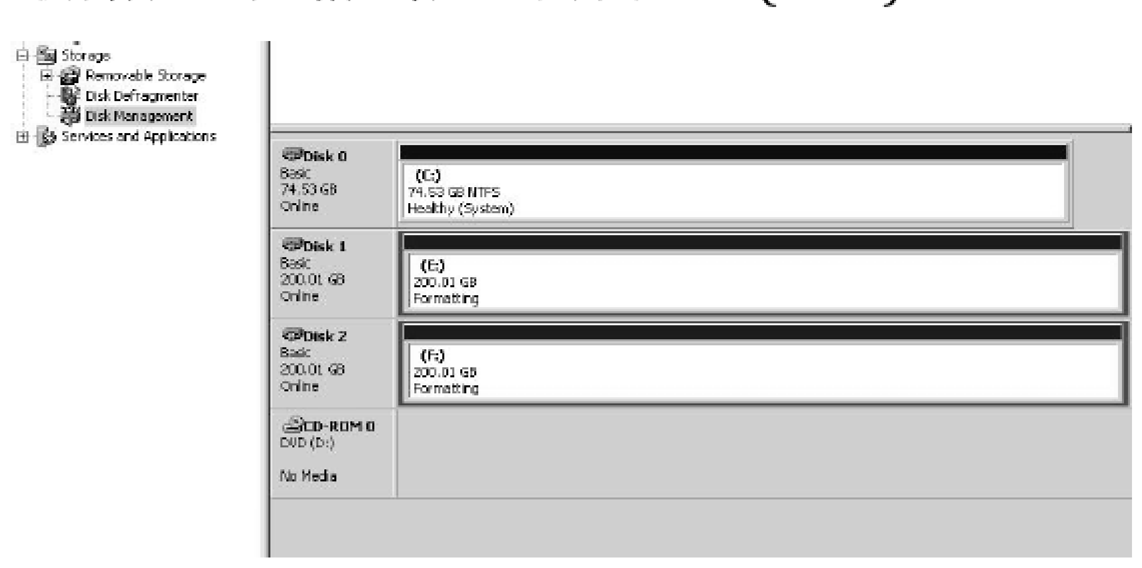


图12.36　初始化之后格式化磁盘

如图 12.37 和图 12.38 所示，这两块磁盘已经可以正常使用了。

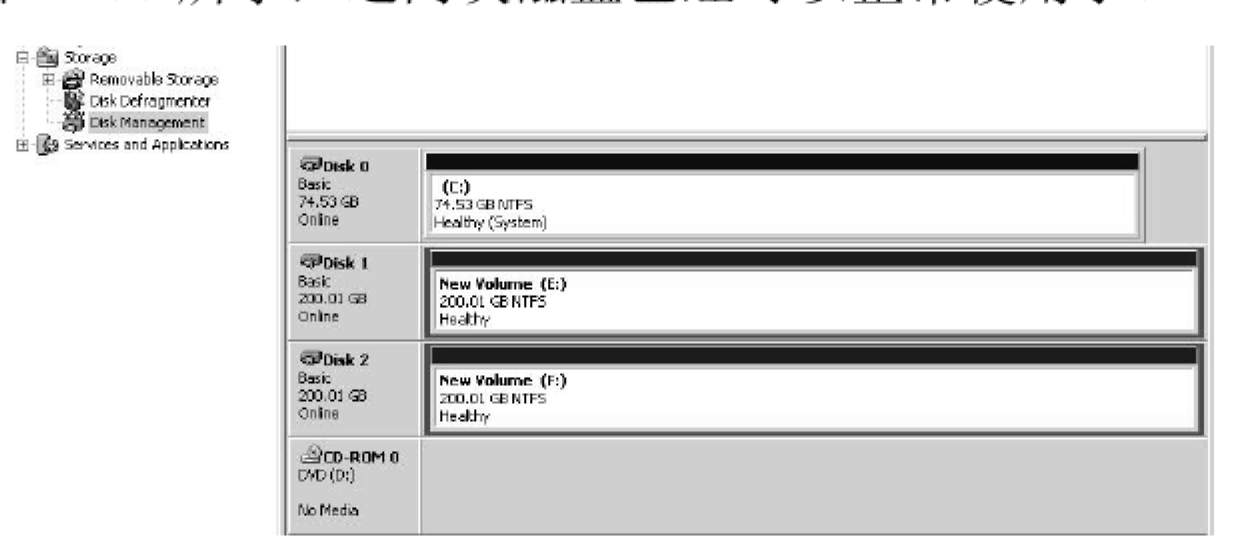


图12.37　磁盘状况正常

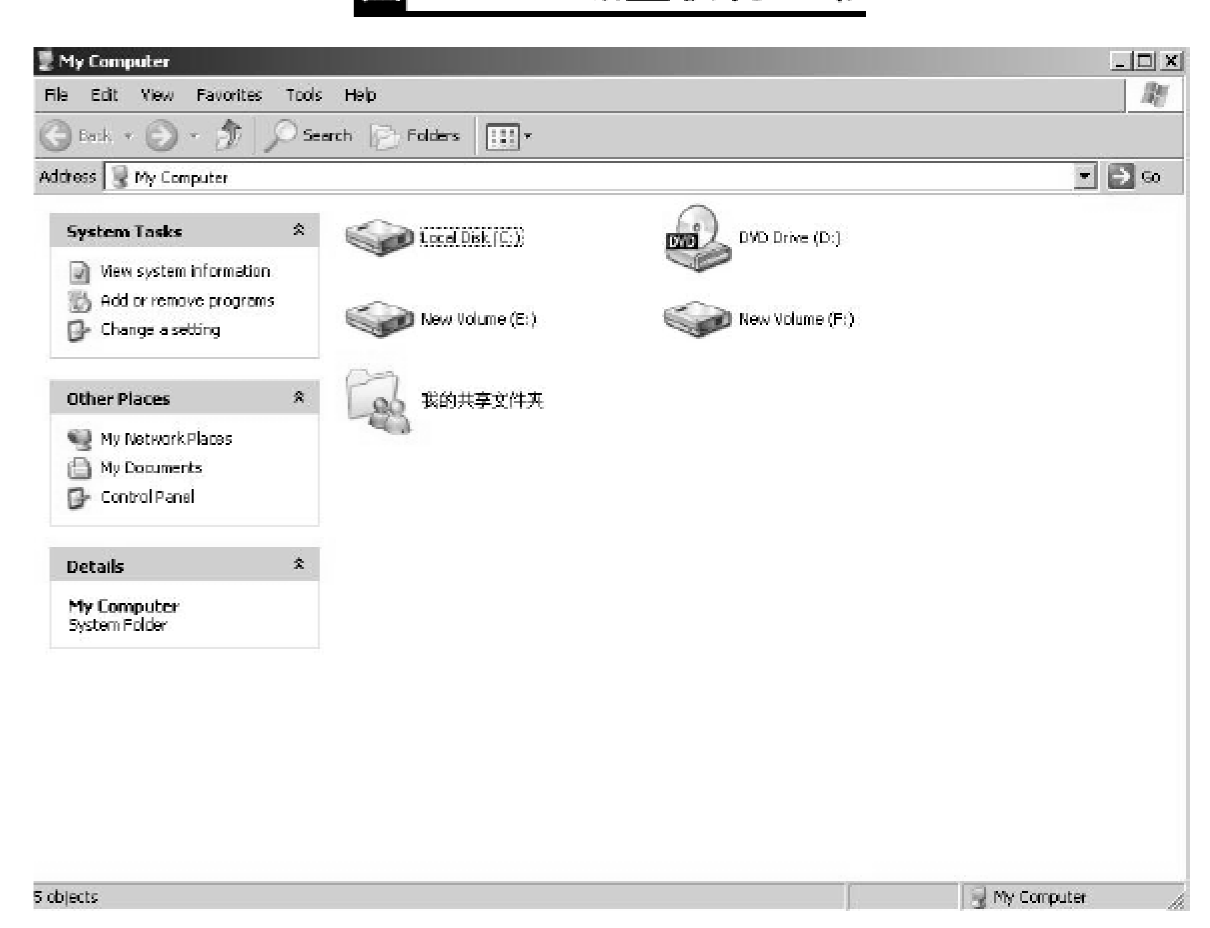


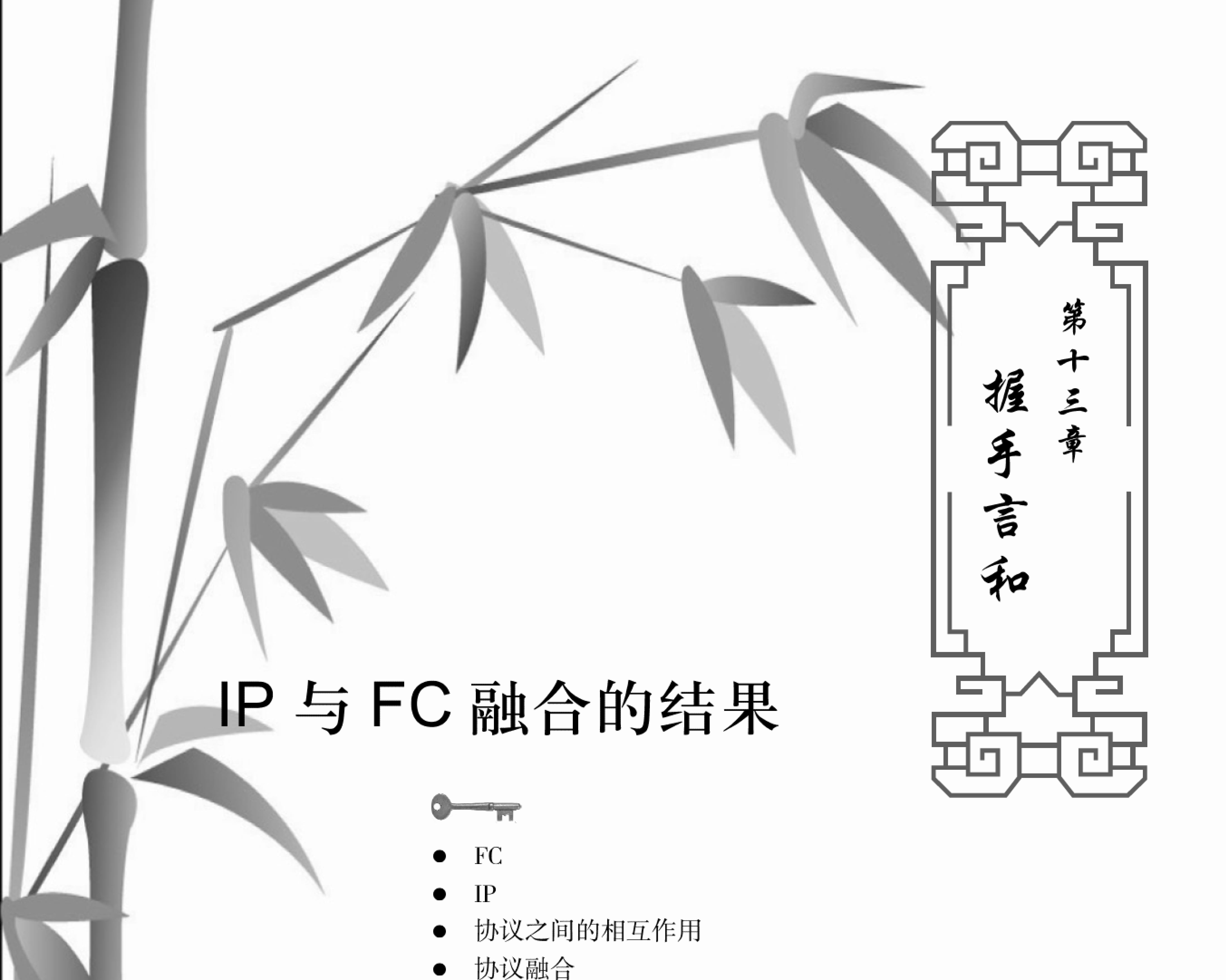
图12.38　“我的电脑”显示的磁盘

12.10 小　　结

IP SAN 出现之后，FC SAN 的统治地位被大大地动摇了。FC 目前唯一可以用来与 IP SAN 抗衡的武器，就是其高速度。但是这个武器的震慑力也在降低。我们可以发现，以太网的速度每次革新都是以 10 倍速为单位，从 10Mb/s 到 100Mb/s、1Gb/s，10Gb/s 速率的以太网也已经发布了。而 FC 每次革新均以 2 倍速为单位，从一开始的 1Gb/s、2Gb/s，到目前正在被广泛使用的 4Gb/s 速率，而 FC 的下一个速率级别为 8Gb/s(或者 10Gb/s)，而且这个标准还正在制定当中。以太网已经在计划上抢先了 FC 一步。不知道在 10Gb/s 时代，FC 还能不能守住它那仅有的一点点沙漠绿洲。



IP SAN 还是 FC SAN？这恐怕真的快要成为一个信仰问题了。



IP 与 FC 融合的结果

- FC
- IP
- 协议之间的相互作用
- 协议融合

话说 FC 和 IP 各占一方，谁也不让谁，互相竞争了数年，两者各立门派，势不两立。但是“夫天下之势，分久必合，合久必分”。

数年来，两者在市场上竞争的可谓你死我活。FC 仅仅拿着 FC SAN 的速度和稳定性来炮轰 IP SAN，而 IP 也不甘示弱，处处举着可扩展性和成本的大旗，声讨 FC SAN，闹得江湖上风风雨雨。FC 凭借着它的速度优势，占据了高端市场，而 IP 则以成本优势在低端市场占据了一席之地。然而两人谁都想一统天下，把对方彻底驱逐出市场，但是，相持数年了，谁也没能把谁干掉。两人都累了，这么多年的互相攻击，谁也没有取得丝毫胜利，FC 还是稳固的占据高端市场，IP 依然驰骋低端。

终于有一天，FC 和 IP 决定握手言和，不再投入无谓的人力、物力、财力来和对方竞争。与其大肆攻击对方，不如多用点精力来提升和发展自身的技术，同时学习对方的技术，取长补短，方为正道啊！！ FC 和 IP 彻夜长谈，终于取得了一致的见解，决定双方各取所长，共同为江湖做贡献。

首先，FC 决定由 IP 入股自己的公司，给 FC SAN 提供更高的扩展性架构解决方案；同时，FC 也入股 IP 的公司，给 IP 提供研发经费，用于其研发出基于以太网的、新型的、适合存储区域网络的专用协议体系。

13.1 FC 的窘境

入股 FC 公司之后，IP 便开始研究如何将 FC 协议体系转向可扩展的、开放的结构。说到可扩展且开放的网络传输协议，一定非 TCP/IP 末属。可是 FC 和 TCP/IP 是完全两套毫不相干的协议体系，如果将 FC 全部转为 TCP/IP，那岂不是叛变成 IP SAN 了么？但是如果丝毫不变，那只能是 FC SAN，还是不具备开放和扩展性。

1. FC 的扩展性问题

FC 为什么扩展性差？就是因为如果通信双方距离太远的话，需要自己架设光缆，或者租用电信的专线光缆，这两者成本都很高。如果租用电信部门的专线光缆，则 FC 最低速度为 1Gb/s，且租用电信部门的 1Gb/s 带宽的专线光缆，其费用不是一般机构能承担的。


提示 目前电信提供的专线接入，其骨干网一般采用 PDH 或者 SDH 协议传输，到终端用户所能承受的速率为 2M 的 E1 线路。当然也可以直接从高速骨干直接分离出相对高速的线路，比如 OC3，OC48 等，但是费用还是过于高昂，无法承受。

E1 线路有自己的编码格式，不能将电信部门接入的光纤直接插到 FC 设备上，因为两端的编码方式不同，不能和局端的设备建立连接，所以需要增加一个协议转换设备(准确来说是协议隧道封装设备)，将 E1 协议解封装，转换成协转设备后面的协议逻辑，比如 V35 串口、以太网等其他协议。目前已经存在 FC over Sonet，FC over ATM 等协转设备了，不过这些专线的扩展性仍然不强，而且这种方案以及对应的设备也非常昂贵和稀少。

目前看来，如果要扩展 FC 网络，让相隔很远的两地之间用上 FC 协议，最好的办法就是自己架设专用光缆，可是自己架设光缆也只能在自己可控的范围内，比如一个大厂区之内，但是如果是在市内，或者两个城市、两个省之间，私自架设光缆是绝对被禁止的。

2. 解决方案

怎么办？首先，要走出去，就一定要租用电信部门的线路。电信提供了两种线路，一种是接到 Internet 的线路，也就是接入电信部门的 Internet 运营网络，通信的双方都接入，并且使用 TCP/IP 通信。另一种，就是光缆专线，也就是通信的双方都接入电信部门的专用传输骨干网络，这条专线端到端的带宽由接入提供商保证，只要两端的设备支持，其上可以运行任何上层协议。上层帧会被底层封装协议(比如 E1 等)再成帧传送到电信部门骨干传输网络中。

虽然 Internet 接入可以获得 100Mb/s 或者 1000Mb/s 的速率，但是这只是本地带宽(从本地到局端设备之间的链路带宽)，端到端的带宽，以现在的电信部门 TCP/IP 网络环境，除非购买接入商的 QOS 或者 MPLS TE 服务，否则没有人能够保证两点间的通路带宽(速率)。

> 提示
> 如果两地之间相距很近，那么不妨考虑 Internet 链路。因为如果两地同时接入相同城市的 ISP 网络，数据包被路由的跳数就不会很高，甚至有可能只经过 1 跳或者 2 跳便可以被对方收到。更有甚者，同城的两地可能连接在局端的同一台设备上。这样可获得的带宽速率就会非常可观，就可以像在内网通信一样利用 VPN 来让两个站点之间联通。但要澄清一点，由于 Internet 链路不能时刻保障稳定的带宽，所以这种方法只适合对数据传输实时性要求不高，但同时又要求高带宽的情况。

而专用线路虽然保证了带宽，但是只能承受 E1 等低速专线，且价格相对 Internet 接入要贵很多。而且目前只有 V35－E1 封装解封设备和 E1－以太网封装解封设备，并没有 E1－FC 封装解封设备。而 V35 串口和以太网这两种二层协议，都普遍被用来承载 IP 协议，所以目前来说，E1 一般用来承载 IP 作为网络层协议。有些路由器自带 E1 封装解封模块，可以不用外接协转，直接连接从光端机分离出来的一路或者几路 G703 或者 BNC 接头，直接编码与解码 E1 协议。但是这些也都是 IP 路由器，和 FC 丝毫没有关系。

可以看出，FC 如果脱离了“后端专用”这四个字到开放领域，显然是无法生存的。而 IP SAN，则软硬通吃，只要有 IP 的地方，不管其下层是什么链路协议，就可以部署 IP SAN。这就是为何称 TCP/IP 为协议中的秦始皇的原因，秦始皇统一了货币，到哪里都通用，同样，TCP/IP 也统一了下层凌乱的各种协议。

13.2 协议融合的迫切性

说到这里，租用 Internet 线路，只能承载 IP，而租用点对点专线，也普遍用来承载 IP，可能感觉 FC 的扩展似乎就是死路一条了。但是，IP 想起了 ISCSI，当初自己不就是把 SCSI 协议给封装到了 TCP/IP 协议中来传输，才扩展了 SCSI 协议么？也就是说如果将一种协议封装到另一种协议中传输，就可以使用另一种协议带来相应的好处了。不妨就这么假设一下，FC 不可扩展，TCP/IP 扩展性很强，那么如果把 FC 协议封装到 TCP/IP 协议中来传输，是不是也可以获得 TCP/IP 的扩展性呢？这个想法比较大胆，因为 FC 本身也是作为一种可以传输其他协议的协议，FC 甚至可以承载 IP，作为 IP 的链路层，那么为什么现在却反过头来需要用 IP 来承载呢？



> Protocol over Protocol，PoP，即一种协议被打包封装或者映射到另一种协议之上。这种思想在网络协议领域中经常使用。我估且称其为“协议融合”，认为其已经可以形成一个独立的科目。

要谈协议融合，还得从以太网和 TCP/IP 说起。

以太网和 TCP/IP——不能不说的故事

前面已经详细介绍了以太网和 TCP/IP 协议。我们知道，以太网是一个网络通信协议。

提示 记得某人曾经说过一句话："网络就是水晶头。"这句话比较有意思，它反映出说这句话的人对网络的不了解，但是也证明他平时所见到的网络，确实只有水晶头，且以太网普遍使用水晶头，那么"网络就是水晶头"这句话，也不是那么可笑了。它从某种角度也反映出了以太网在当今的普及程度。

前面讲到以太网是可以寻址的，也就是说它涉及了 OSI 第三层网络层的内容。大家都连接到一个以太网环境中，不需要任何其他上层协议，就可以区分开对方，进行通信。既然这样，为什么连新闻联播的主持人都知道 Internet 是利用 TCP/IP 协议而不是以太网来通信呢？为何我们总是说以太网+TCP/IP 协议二元组，而不是仅仅说以太网，或者 TCP/IP 协议？

因为以太网和 TCP/IP 协议是逻辑上分开的，它们各自是不同的协议体系，那么为什么总是把他们组合起来说呢？它们之间有什么割舍不断的恩恩怨怨呢？这其中原因，还要从 IP 讲起。

1. IP 本位

前面也说过了，IP 就是一个身份标志，一个用来与其他人区别的一个 ID。以太网协议中规定的 MAC 地址，从原理上讲，就足够用来区分网络中各个节点了。但是前面也分析过，完全靠 MAC 来寻址的缺点：一是 MAC 地址太长，48bit，用于路由寻址时效率太低；二是世界上并不是每个环境中都用以太网来建立网络的，除了以太网，还有其他各种方式的网络系统，各自有各自的寻址方式，如果要让所有类型的网络之间无障碍的相互通信，就需要一个秦始皇来统一天下的货币。

IP 就是这个被选中的货币。不管以太网，或者串口协议，或者 FDDI 等类的局域网方式，我们最终都要让其之间相互通信，才能形成 Internet。



如果你是秦始皇，你会怎么来处理各国众多的货币呢？虽然秦始皇最终将其他货币回收废除了，但是 IP 却不能在短时间内将所有网络形式都废除，而用以太网统一，因为现在已经不是一个人就说了算的时代了。秦始皇可以在各个使用不同货币的地方设立一个专门的兑换机构，只要到了这个地方，就兑换成这里使用的货币。

同样，我们也给每个网络设立一个网络地址兑换设备，也就是协议，将统一的 IP 地址兑换成这个网络的自用私有地址，用这种方式实现各种类型网络的相互联通。网络中的兑换机制，是通过 ARP 协议实现的，ARP 协议可以将一种网络地址映射成另一种网络地址。每种网络要想用 IP 来统一，都必须运行各自的 ARP 协议，比如以太网中的 ARP 协议，帧中继网络中的 ARP 协议等。

对于以太网来说，IP 就是统一货币，MAC 就是以太网货币。另外，还有各种各样其他类型的货币，比如主机名(Hostname)、域名等。大家在访问网站的时候，其实就是和提供网站服务的服务器来建立通信，获取它的网页和其他服务，在 IE 浏览器中输入这个网站的域名之后，DNS 兑换程序会自动向 DNS 服务器查询，获得这个域名所对应的 IP 地址，然后

用 IP 地址与服务器通信。

数据包带着 IP 地址到了服务器所在的局域网之后，会通过局域网的路由器发出 ARP 请求，来把 IP 地址再兑换成服务器所在局域网络的地址，如果服务器所在的局域网是以太网，则对应成 MAC 地址，然后通过以太网交换设备，找到这个 MAC 地址所在的交换机端口，将数据包发向这个端口，从而被服务器收到。

为什么要经过多次兑换呢？首先把 IP 转换成域名，是为了方便记忆，不必记忆那些复杂的 IP 地址。其次把 MAC 转换为 IP，是为了天下统一，相互流通。

其实如果所有人都用以太网联网，那么就可以完全抛弃 IP 这一层寻址了，但是实际是不可能的，以太网现在还没有一统天下，而且就算一统天下了，人们也似乎不愿意抛弃 IP，就像在同一个局域网内，还是用 IP 来直接通信，而不是直接用 MAC。TCP/IP 实在是被使用的已经太普遍了，以至于就算牺牲一点性能，局域网内通信也普遍使用 IP。而实际上，以太局域网内部通信的话，NetBEUI 协议的性能比 TCP/IP 协议要高许多。

其实整个 Internet，不仅仅都是以太网，以太网适合局域网联网通信，但是不适合广域网情况，广域网的联网协议，比如 PPP，HDLC，Frame Relay，x25，ATM 等，也像以太网一样各有各的寻址体系。在一个 Internet 上有这么多种不同地址的网络，它们之间若要相互融合、寻址，就必须在各种地址之间，相互翻译、转换、映射，数据包每经过一种网络，就转换一次，这样非常麻烦。IP 地址的出现使得所有联网的节点，不管用的是以太网，还是 Frame Relay，统统都分配一个 IP 地址给每个节点，对外最终以 IP 地址作为寻址地址，而将 IP 地址再映射到自己所在网络的所使用的地址上，比如 IP 映射到以太网的 MAC，或者 IP 映射到 Frame Relay 的 DLCI，映射到 ATM 的地址等。

用来进行地址映射的程序，称为 Address Resolution Protocol，即 ARP。很多人听到 ARP，就认为是以太网，其实这也是错误的，ARP 不仅仅代表以太网中的 IP 地址和 MAC 地址的映射，它代表任何种类地址之间的映射对应关系，从这一点来说，DNS 协议也应该归入广义的 ARP 协议中。

IP 统治了 OSI 的第三层，将原来占据第三层的凌乱地址种类统一了。映射到(承载于)以太网的 IP，称为 IPoE(IPoE 也就是“基于以太网的 TCP/IP”)；映射到帧中继的 IP，称为 IpoFR；映射到 ATM 的 IP，称为 IPoA 等。从此一种新的概念诞生了：PoP，即 Protocol over Protocol。

2. IP 缺乏传输保障功能

IP 统一了天下还不够，因为 IP 最大的作用就是寻址和路由以及适配链路层 MTU，它并不提供其他功能，而作为一个健全的网络传输协议，必须具有传输保障功能。而以太网是一个面向无连接的网络，它不保障数据一定会传送的对方，是一个不负责任的网络，不管目的端口有没有收到，源端口只管向外发送。而 Frame Relay 协议，其前身 x25 协议，是一个有着很好传输保障功能的协议，在 TCP/IP 没有出现之前，x25 的传输保障机制做得非常到位，因为 x25 的设计初衷，就是为了运行到极其不稳定的链路上。而随着链路质量的不断提高，x25 的做法显得越来越因噎废食了，所以其改良版本 Frame Relay，就逐渐替代了 x25，FR 抛弃了 x25 中很多无谓的传输保障机制，而仅仅留下一些流控机制。相对于以太网的不负责任，FR 起码在链路层面，实现了比较好的流控措施。

但是，不管是以太网，还是 FR，都没有实现端到端的传输保障。端到端，是相对于“过路”来说的。过路是指在两个终端之间通信路径上的网络设备之间的路径。链路层的传输保障就是一种过路保障，因为链路层只保证相连的两个设备之间传送数据正常无误，但是不能保障通信最终端接收和发送的数据正常无误。因为在一个典型的包交换网络中，数据包一般都是一跳一跳的被传送的，每一跳两端的设备用链路层协议进行传输保障。

但是最终目的是要让通信的最终两端无误的收到数据，才能算作真正的传输保障，即端到端的保障。而 FR 协议所做的，只是在过路的时候保障链路正确传输。如果链路正确传输给了终端，而终端到最终上层的某个环节出错了，那么数据同样也是错误的，所以，要实现端到端的传输保障，一定要在最终传输终端上运行一个侦错和纠错逻辑，用来发现链路层所发现不了的错误。图 13.1 为端到端保障与过路保障的示意图。

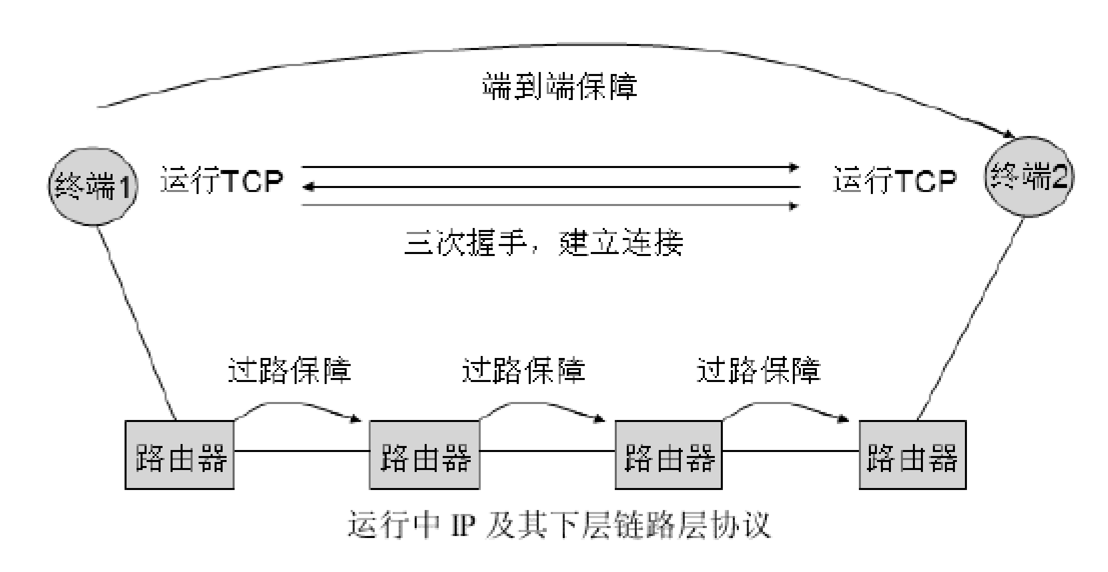


图 13.1　过路保障与端到端的保障

3. TCP 保驾护航

为了实现这个目的，TCP 出现了。TCP 作为一个程序运行在通信的两个终点，不管两点之间用什么样的链路连接，经过了多少网络设备，TCP 程序始终运行在通信终端上，监控终端最终发送和接收到的数据包的顺序、缓存区、校验等信息，检查是否出现丢包、阻塞等事件，一旦发现错误，立刻纠正重发数据包。

TCP 不是运行在通信路径上的，而是运行在通信终点的两端设备上。即使过路链路保障机制再健全，TCP 也是有必要的，因为数据包只有被终端正确接收到，才能算真正的传输保障。

所以，在 IP 之上，又凌驾了一层 TCP 逻辑，用来保障端到端的无误传输。而 FR 等链路层协议的保障机制，只能保障本段链路传输无误，不能保障端到端的正确收发，所以只能沦为数据链路层协议的角色了，用来承载 IP 和 TCP。

我们可以体会到，协议之间也是在互相利用、互相排挤、吞并，融合，以适应不同的应用环境，因为不可能为每一种应用环境都设计一种协议，协议之间互相利用、融合，才是最好的解决办法。

4. 最佳拍档——TCP/IP 和以太网

现在可以回答上面没有找到答案的那个问题了，为什么以太网偏要和 TCP/IP 组合成一对呢？因为以太网使用得太广泛了，而 OSI 的第三层、第四层，也几乎被 IP、TCP 给统一了，所以以太网+TCP/IP，当然就成了一对好搭档了。

虽然一个协议可能实现 OSI 的所有 7 个层次，但是如果它要和其他协议合作，那么就要有个分工，而不能越权，比如 IPoA，ATM 只要传输 IP 包到目的就可以，而不管数据是

否出错、乱序等，虽然 ATM 可能有这个功能。以太网虽然自己可以寻址，但是它还是配合 IP，进行 IP 到 MAC 的映射，统一使用 IP 寻址，它默默无闻，所有光辉都被 TCP/IP 所披挂。

13.3 网络通信协议的四级结构

网络通信协议，一般可以分成 Payload 层、信息表示层、交互逻辑层和寻址层。其中最重要的是交互逻辑层，它是一个协议的灵魂。

1. Payload 层

Payload 是协议所承载的与本协议逻辑无关的最终数据，是通信终端通过本协议最终需要传送给对方的数据。Payload 也就是协议所运输的货物。Payload 层中的数据，既可以是最终应用产生的数据，也可以是另一种协议的信息表示层+Payload 数据。如果 Payload 封装的是最终应用产生的数据，则表示这个协议是直接被上层应用程序来调用，从而完成程序之间的远程网络通信的。

如果 Payload 封装的是另一种协议的信息表示层+Payload 数据，那么就证明这个协议此时正在承载那个协议。比如协议 A 封装了协议 B 的信息表示层+Payload，则就可以说协议 A 封装了协议 B，或者协议 A 承载了协议 B，或者说协议 B is over 协议 A(BoA)。我们后面会描述一种协议被 Map(映射)到另一种协议，而不是被封装，这种融合方式称为 AmB，是彻底的协议转换，而不是仅仅做隧道封装。

2. 信息表示层

信息表示层就是附加在 Payload 数据之外的一段数据，也称作协议开销，因为这段数据和最终应用程序无关，是运行在通信双方的通信协议用来交互各自的状态，从而使双方作出正确动作的一段重要数据。这段数据可以想象成提货单或者信封。信封封装了信纸，信封上的地址、姓名等信息，就是信息表示层，它可以让对方检测到当前通信所处的状态。

3. 交互逻辑层

这一层其实就是运行在通信双方协议系统上的动作程序代码逻辑，它根据对方传送过来的信息表示层数据来作出相应的动作逻辑，再生成自己的信息表示层发送给对方，然后对方再做相同的处理判断动作，就这样完成通信双方之间的正确动作。交互逻辑层其实就是协议的设计思想。交互逻辑层对于每种协议都不相同，但是很多都类似，可以说网络通信协议基本思想是类似的，因为它们所实现的目的都是一样的，就是将数据通过网络传输到目的地。

正因为如此，各种协议的交互逻辑层才可以相互融会贯通，将一种协议的逻辑，映射翻译到另一种协议的逻辑，从而将各种协议的优点结合起来，完成目标。协议逻辑层一般都是运行在通信双方两端的，但是像 IP 路由协议等，通信双方经过的路径上的所有设备，也都需要运行，因为 IP 包是一跳一跳被接收并且转发的。

4. 寻址层

它是帮助协议来找到需要通信的目标的一套编址和寻址机制。比如 IP 地址、MAC 地址、

DLCI 地址、电话号码等。如果是点对点传输协议，则可以忽略此层，因为不需要寻址。而且不同协议之间的寻址层，可以互相映射翻译。

以上的这四层，是任何一个网络通信协议所必须具备的，不管多么简单或者多么复杂的协议。

5. 通信协议的相似性

相似性是通信协议之间相互融合的一个条件。而协议之间相互融合的另一个促成因素，就是协议使用广泛程度不同，有时如果要完成一个目标，不得不借用某种协议。

就像 TCP/IP 协议，TCP/IP 协议占领了全球 Internet 的领地。如果有一种协议想跨越地域或国家来进行通信，但是自己又无能为力，因为它首先没有专门为它准备的物理线路，其次它的设计，也就不适合大范围、长距离的广域网环境，那么它只能来租用 TCP/IP 协议，将自己封装到 IP 包中传送。能适合 Internet 规模的网络通信协议，唯 TCP/IP 莫属！而其他协议想要完成 Internet 范围的通信，就不得不借助 TCP/IP，搭 TCP/IP 的车，让 TCP/IP 来承载它们。它们是怎么搭上 TCP/IP 的快车呢？

我们不妨类比一下。在整理本章的时候，恰逢大连刚刚开通了一艘新的火车箱滚装船。我想用这个例子来比喻协议融合，再适合不过了。从山东烟台到大连，最近的路径就是走渤海湾水路，如果搭乘陆路火车，则需要绕一大圈，所以很多货运汽车，甚至火车，都选择乘船到大连，下船后，车厢用火车头拉走，这样，在增加很少成本的条件下，节约了大量时间。协议融合同样遵循这个原则，只要能使总体拥有成本降低，性价比提高，任何协议都可以融合。

13.4　协议融合的三种方式

协议和协议之间的相互作用，有三种基本的思想。

- 第一种是调用(Use)，也就是一种协议完全利用另一种协议。
- 第二种是隧道封装(Tunnel)，一种协议将另一种协议的完整数据包全打包遂道封装到新协议数据包中。
- 第三种是映射(Map)，也就是一种协议对另一种协议进行映射翻译，只将原来协议的 Payload 层数据提取出来，重新打包到新协议数据包中。

1. 调用关系

所谓调用，也就是一种协议自身没有某些功能，需要使用另一种协议提供的功能。比如 TCP 调用 IP，因为 TCP 没有寻址功能，所以它利用 IP 来寻址。而 IP 又可以调用以太网，因为 IP 只是一个寻址功能，它没有链路传输的功能，所以它利用以太网提供的链路传输(交换机、Hub 等)。IP 调用 PPP 来传输等，也就是上层协议为了达到通信目的，使用另一种协议为其服务。这种关系严格来说，不算是融合。

2. 隧道关系

隧道封装，顾名思义，就是将一种协议的完整数据包(包括 Payload 和协议开销)作为另一种协议的 Payload 来进行封装，打包传输到目的地，然后解开外层协议的封装信息，露出

内部被封装承载的协议完整数据包，再提交给内层协议处理逻辑模块进行处理。也就是说，进行协议转换的设备根本就不需要去理解内层协议到底是什么东西，到底想要干什么，只要将数据包统统打包发出去。Tunnel 的出现，往往是由于被 Tunnel 的协议虽然和外层协议都在某一方面具有相似甚至相同的功能，但是在某些特定的条件下，被 Tunnel 协议不比外层协议表现得优秀，不适合某种特定的环境，而这种环境，恰恰被外层协议所适合。这就像用船来装火车箱一样。Tunnel 的另一个目的是伪装内层协议。

3. 映射关系

Map 是比 Tunnel 更复杂、更彻底的协议融合方式。所谓 Map，也就是映射，就是将内层协议的部分或者全部逻辑，映射翻译到外层协议对应的功能相似的逻辑上，而不是仅仅做简单的封装。Map 相对于 Tunnel，是内外层协议的一种最彻底的融合，它将两种协议的优点，融合得天衣无缝。内层协议的 Payload 层在 Map 动作中是不会改动的，因为 Payload 层的数据只有两端通信的应用程序才能理解。

13.5 Tunnel 和 Map 融合方式各论

例如火车、汽车是两种运输工具，它们看似有太多的不同，但是它们的目的都是相同的，都是为了将货物运送的目的地。而火车需要跑在铁道上，但汽车需要跑在公路上(物理层不同，链路层不同)；火车因为铁轨很平滑，需要用钢铁轮子，而汽车因为公路很颠簸，需要用充气轮胎；火车不需要红绿灯来制约，而汽车跑在公路上，会有很多红绿灯来制约它；火车由于跑在专用的铁轨上，所以它能达到很高的时速，而汽车由于跑在共享的公路上，它能，但是不敢达到太高的时速，火车只能按照它的轨道来运行，而汽车几乎随处可去……

以上列举出了火车和汽车的种种特点，相应的飞机、轮船、火箭等都可以拿来对比，这些特点就像各种通信协议自身的特点一样。同样都是运输货物，但是它们都适应了不同的需要。只不过网络通信协议运输的不是货物，而是一串 0 和 1，是高低变化的电平，是数据，是信息。不同的通信协议同样也是为了满足不同的情况、不同的需求。TCP/IP 协议满足了 Internet 范围的网络通信；FC 协议满足了后端存储的专用高速公路这个环境，二者都各自占有自己的领地，谁也取代不了谁。就像铁路不可能为了和民航竞争，而把轨道往天上修，航空公司也不可能为了和陆运公司竞争，而让飞机跑在公路上。

TCP/IP 适合整个 Internet 范围的通信，而 SCSI 协议不适合，所以如果 SCSI 协议需要跨越大范围通信，就要将其承载到 TCP/IP 上，也就形成了 ISCSI 协议，然而 TCP/IP 根本就不关心什么是 SCSI，更不知道 SCSI 是怎样一种作用逻辑，它只是负责封装并传输。同样，因为以太网是个面向无连接的网络，没有握手过程，也没有必要有终端认证机制、没有 NCP 机制(PPP 协议中用来协商上层协议参数的机制)，而 PPP 却有这些机制，它非常适合 ISP 用来对接入终端进行认证和管理，但是 PPP 使用程度远远不如以太网广泛，怎么办？融合吧！于是形成了 PPPoE 协议。

13.5.1 Tunnel 方式

ISCSI 和 PPPoE 这两个协议，是典型的 Tunnel 模式。前面已经给 Tunnel 下过定义了。首先一种 PoP 的模式被定义为 Tunnel 的前提，就是这两种协议对某一特定的功能，均有自己的实现。如果一种协议在某方面的功能，另一种协议没有实现，那么另一种协议就是“调用”那种协议，而不是被 Tunnel 到那种协议。比如，IPoE 就是典型的调用，而不是 Tunnel 或者 Map，因为 IP 没有链路层功能。



IP 与 Ethernet 之间的编址逻辑是映射关系而不是使用关系，即 IP 地址与 MAC 地址的相互映射。

用 ISCSI 来分析，TCP/IP 可以实现寻址和传输保障，SCSI 协议也可以实现寻址和传输保障，所以它们具备了这个前提；同样 PPPoE 也是一种 Tunnel 方式的融合协议，因为 PPP 和 Ethernet 都是链路层协议。

1. VPN 的引入

Tunnel 的另一个作用，就是伪装。有时候虽然两种协议实现的功能、适用环境都相同，但还是将其中一种 Tunnel 到另一种之上,这是为什么呢？有些情况确实需要这种实现方式。比如 IP 协议中的 GRE，通用路由封装，就是这样一种协议。它将 IP 协议承载到 IP 协议本身之上，自己承载自己，再封装一层，这样就可以使得一些不能在公网路由的 IP 包，封装到可以在公网路由的 IP 包之中，到达目的后再解开封装，露出原来的 IP 包，再次路由。这就是伪装。

利用这种思想，人们设计出了 VPN，即 Virtual Private Network，用来将相隔千里的两个内部网络，通过 Internet 连接起来，两端就像在一个内网一样，经过 Internet 的时候，使用公网地址封装内网的 IP 包。这是最简单的 VPN。在这基础上，又可以对 IP 包进行加密、反修改等，形成 IPSEC 体系，将其和原始的 VPN 结合，形成了带加密和反修改的 IPSEC VPN，真正使得这种 PoP，穿越外层协议的时候，能够保障数据安全。

2. 例解 Tunnel

下面再举个例子来说明，到底什么是 Tunnel。

邮政系统，目前已经是举步维艰。21 世纪之前，网络还没有很普及，除了电话、电报，写信似乎是大家长距离通信的唯一选择。寄信人将自己的信件(数据，Payload)装入信封(协议信息表示层数据段)，填好收信人地址、邮编、名称(通信协议的信息表示层、寻址层)等，交给邮局(网络交换路由设备)，由邮局进行层层路由转发，最终到达目的地。

IP 网络和邮政系统极其相似。而为什么邮政系统目前已经陷入了困境呢？原因就是竞争。

进入 21 世纪之后，物流业快速兴起，它们借助陆路、水路、航路、铁路等 “链路层”，加上自己的一套流程体系(协议交互逻辑)，充分利用这些资源达到物流目的。以前只有邮政一种方式，而现在物流公司多如牛毛，每个公司都有自己不同的物流体系，但是基本思想

大同小异，都是要将用户的货物运送到目的地。

21 世纪，虽然网络已经很发达，但是网络只能走信息流，走不了实物流。所以物流公司还是能占据一定市场。



我们来看看 21 世纪，用户是怎么来寄出一封信件或者包裹的。同样寄出一封信，如果还是用古老的协议，比如信封 + 80 分邮票的形式，还是可以的，大街上现在还有邮筒。但是很多快递公司，也提供信件包裹服务，只不过他们用的信封，比普通信封大，结实，而且他们信封上的标签，所包含的信息更加具体和丰富，比如增加了收件人电话、发件日期、受理人签字、委托人签字等。邮政信封具有的，快递信封都具有。

这样就可以看出这两种协议的不同之处了。用户可以把信件封装到邮政普通信封直接发送，也可以封装到快递公司信封中发送，也就是选用其中一种协议。

那么如果用户先把信件(最终数据)封装到普通信封中，填好信封头信息(协议信息表示层和寻址层)，然后将封装好的普通信封，再封装到快递公司的信封中，并再次填一份快递公司的信封头信息。快递公司按照这些信息，将信件送到目的地，目的收到之后，解开外层信封，然后解读内层信封的信息头，再次转发，或者直接打开。刚才描述的这种情况，就是一个典型的协议 Tunnel 方式的相互作用，把邮政协议，Tunnel 到快递公司的协议，这种 Tunnel 的目的，就是为了获得快速、优质的服务，因为普通邮政协议提供不了快速高效的服务。



我们再来看这种情况，比如快递公司 A，在北京没有自己的送货机构，但是青岛有人需要向北京送货，怎么办？

此时当然要考虑借助在北京有送货机构的快递公司 B，让他们代送，将信件封装到快递公司 A 的信封，然后再将 A 的信封装入快递公司 B 的信封，让快递公司 B 做转发，到目的地之后，B 的送货员剥开外层信封，最终用户会收到一个快递公司 A 的信封，客户就认为是快递公司 A 全程护送过来的，其实不是。这样就很好地伪装了信件。这是 Tunnel 的另一个目的。

13.5.2 Map 方式

说完了 Tunnel，我们再来说说 Map。Map 就是将一种协议的逻辑，翻译映射成另一种协议的逻辑，Payload 数据完全不变，达到两种协议部分或者完全融合。

还是快递公司的例子。两个快递公司(两种协议)，快递公司 A 在青岛没有自己的送货机构，但是 B 有。所以 A 和 B 达成协议，A 将青岛地区的送货外包给 B，凡是 A 公司在青岛的业务，都由 B 来运送，但是表面上必须保持 A 的原样，这种方式目前实际已经广泛使用。起初的做法是：先将客户信件装入 A 信封，然后再封装一层 B 信封，带着 A 信封来转发，也就是 Tunnel。后来，B 公司嫌这种方法浪费了成本，因为额外携带了一个 A 信封，这增

加了信件的重量和信封成本。所以 B 公司琢磨出一套方法。

1】 先让 B 公司的取件人了解寄件人所要提供的信息，此时取件人担当 A 公司的角色，用户认为取件人是 A 公司的，用户按照 A 公司的协议，将信封头信息告诉取件人；

2】 然后取件人此时并没有将信件装入 A 公司信封，而是直接装入了 B 公司信封，但是在填写 B 公司信封头的时候，取件人将用户提供的针对 A 公司特有的信封头信息，转换翻译成 B 公司特有的信封头信息；

3】 经过 B 公司转发后，到达目的地之后，送货员再次将 B 公司的信封头信息，转换翻译成 A 公司所特有的信封头信息。

这样，两端的用户，同样也丝毫感觉不出中间环节其实是 B 公司完成的。但是这种方式相对于 Tunnel 方式的确节约了 B 公司的成本，使得开销变小了，提高了转发效率。这种方式的协议之间的相互作用，就是 Map。

1. IP 和以太网之间的寻址关系 Map

最简单的 Map 就是 IP 和以太网之间的寻址关系 Map。IP 地址必须映射到 MAC 地址，才能享受以太网的服务。正如 IP 和以太网之间的 Use+Map 关系一样，实际上，各种协议之间的相互作用，不可能只是其中一种作用方式，寻址体系之间一定需要 Map(同种协议自身 Tunnel 的情况除外)，交互逻辑层可以 Tunnel，也可以 Map，Payload 一定需要 Tunnel。所以针对协议不同的层次，都有相对应的相互作用方式。

2. 协议交互逻辑的 Map

协议交互逻辑的 Map，比寻址层的 Map 要复杂得多。寻址层的 Map 只要维护一张映射表就可以，交互逻辑的 Map 则需要维护一个代码转换逻辑模块。

两种协议的状态机的互相融合作用是很复杂的。比如 TCP 的流控机制和 FC 协议的流控机制之间的 Map，TCP 是靠窗口机制实现端到端的流控，FC 靠 Buffer to Buffer(过路流控)和 End to End(端到端流控)两种机制实现流控。如果把 FC 协议承载到 TCP/IP 协议之上，那么就会出现 Tunnel 模式和 Map 模式，当然 Tunnel 中也可能需要 Map，Map 中也同样需要一定的 Tunnel 成分。

我们不妨称作：以 Tunnel 为主的模式和以 Map 为主的模式。

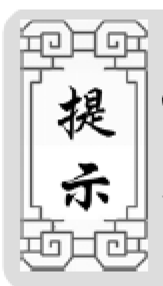
如果是 Tunnel 为主的模式，那么 TCP/IP 根本不管 FC 协议的交互逻辑是怎么样的，TCP 仅仅把 FC 当成 Payload 来封装并传送。

而 Map 模式中，进行 Map 操作的设备或者软件，就需要既了解 TCP/IP 协议的交互逻辑，又了解 FC 协议的交互逻辑，因为只有了解了双方的逻辑，才有可能进行 Map。比如，FC 协议发出了一个信号，说本方缓存将满，请降低发送速度。Map 设备收到这个信号之后，就会 Map 成 TCP/IP 可识别的信号，即本方处理受阻，窗口减小至某某数值，这就是 FC 协议到 TCP/IP 协议关于流控机制 Map 的一个方法。

如果在 Tunnel 模式中，FC 协议发出的这个流控信号，则会被 TCP/IP 给 Tunnel 传送到对方，然后再由对方的 FC 协议模块来根据这个信号来判断流控机制应该做出的动作，动态调整发送速率。

注意：这个信号是直接原封不动的被传送到 FC 协议的对端处理机上处理，而不是像 Map 模式中在本地就终结了 FC 逻辑。Tunnel 模式中，TCP/IP 不参与任何 FC 协议内部的逻辑。

除了 FC 流控逻辑的映射，其他 Flogin 登录机制、连接机制等映射，也都有自己的实现。比如，FC 发起一个 Plogin 过程，那么 Map 设备可以 Map 到 TCP/IP 的一个握手过程等。



Tunnel 和 Map 这两种模式，在第 8 章还有一个将 FC al 的环接入 FC Fabric 中的例子。

13.6 FC 与 IP 协议之间的融合

哗啦……，早晨的微风把 IP 从美梦中吹醒。原来 IP 做了一场美梦。根据梦中的指示，IP 鬼使神差的将 FC 协议映射到了 IP 上。并做了两种模式，一种是以 Tunnel 为主的模式，称作 FCIP；另一种是以 Map 为主的模式，称作 IFCP。

在 FCIP 模式中，通信的双方各增加一个 FCIP 网关，任何 FC 协议的逻辑，哪怕是一个小小的 ACK 帧，都需要封装到 TCP/IP 协议中传输。两端的 FC 协议处理机不会感知到中间 TCP/IP 的存在，它们认为对方就是一个纯粹的 FC 设备。

在 IFCP 模式中，通信的双方各增加一个 IFCP 网关，作为协议转换设备使用。IFCP GW 将 FC 协议终止在本地，提取 Payload 数据，对外以 TCP/IP 设备的形式出现并传输数据，到达对方之后，对方的 IFCP GW 再从 IP 包中提取出 Payload，然后将其封装到 FC 帧中，对其内部以 FC 设备的形式出现。通信双方中间的 TCP/IP 协议，将大部分或者全部 FC 的逻辑都映射成 TCP/IP 的逻辑。

比如每当一个 FC 设备需要和远端的 FC 设备通信，发起 Plogin，那么 IFCP GW 就向对方建立一条 TCP 连接，用多条 TCP 连接和不同的 IP 地址来区分不同的 FC 设备。此外，还需要保存一个 TCP 端口或者 IP 地址对 FC 设备 24bit 的 Fabric 地址的映射表。如果两端的 FC 设备的 ID 有冲突，这个映射表还需要考虑 NAT，将地址翻译成其他 ID。相对于 IFCP，FCIP 协议则不能识别 FC 的逻辑，因为它只是 Tunnel，如果两端 Fabric 中有 ID 冲突的，那么也只能冲突着了。

至此，FC 协议终于可以享受 TCP/IP 带来的扩展性了，FC 搭上了 TCP/IP 的车，远隔千里都可以跑上 FC 协议了。IP 大获成功！IP 和 FC 从此握手言和！

伟大的 SCSI 协议

可以说整个网络存储系统，都起源于一个协议体系，这个协议体系就是 SCSI 协议。网络存储的任何内容，最终都是为了将这个协议体系发扬光大。人们将这个协议强行划分解体成了多个层次，然后把它的最上面的几层，与另一个协议体系——Fabre Channel 协议的下几层进行融合，形成了 FCP 协议，这种协议目前运行在各个厂家的高端磁盘阵列上。还有曾经一度时间，以太网甚至也看好了 SCSI 协议，想与其融合成所谓的“ESCSI”协议，但结果没有成功。以太网失败之后，它的好兄弟 IP 接着上，最终成功地与 SCSI 协议进行

了融合，生成了ISCSI协议，目前也被广泛应用于一些低端盘阵。

为何不是IATA或者FATA呢？原因就是因为SCSI协议体系本身就比ATA协议体系高效并且功能强大，此外，SCSI的硬盘性能也普遍比ATA硬盘转速快性能高，用于服务器系统，所以SCSI当然是首选了。另外，一个巴掌拍不响，SCSI协议本身就想把自己给“嫁”出去，因为它很早就已经迫不及待地将自己分成了很多层次，来吸引其他协议。

协议融合的结果，就形成了目前形形色色的网络存储世界，各种融合协议，各种产品，各种解决方案，好不热闹！而原本的SCSI协议，除了一些磁带机以及主机本机硬盘外，已经不再使用。SCSI融合入了各种协议中，它无处不在，虽然它的躯体已经是七零八落，但是它的精深思想，以及为技术而献身的精神，将在形形色色的技术中永放光芒！！

13.7　无处不在的协议融合

之所以提出“协议融合”这个名词，而不是“协议映射”或者“协议隧道”，是因为“融合”这个词更加通俗易懂；另外，也更加具有生物学色彩。计算机就是人类所创造的另一种形式的“生物”，人类就是计算机的上帝。

1. 协议融合和基因融合

分子生物学家们将不同功能的基因段整合到一起，然后用核糖体蛋白机器读取其代码，表达成肽链，然后折叠成三维结构的新功能蛋白质分子，比如抗冻小麦、发光的白鼠等。这就是基因融合。这个过程与协议融合类似。

协议融合是无处不在的，正如不同快递公司之间的合作一样。甚至连劳动合同方面都出现了融合，劳务派遣公司与劳动者签订合同，然后将劳动者输送到用工单位工作，用工单位不必维护人事系统，将人事系统外包给劳务派遣公司。

2. 航空公司的协议融合

目前，国际上的大多数的大型航空公司都利用IBM或者Unisys的大型机系统作为订票和离港系统的处理机。世界各地的售票和离港终端都通过某种网络系统与大型机连接并且通信。航空业的大机与终端通信协议也经历了纯种和融合阶段。

IBM利用ALC协议与其终端通信，Unisys主机则通过UTS协议与其终端通信。但是随着IP网络的成本不断降低，质量不断提高，UTS和ALC这两种古老的纯种协议，不得不考虑将自己嫁给IP网络，从而出现了MATIP协议，也就是将这些协议承载于IP之上。Cisco公司也为航空业专门开发了这种融合协议，称为ALPS协议。然而ALPS最终没有成为RFC标准，而MATIP协议，却最终登上了RFC宝座。MATIP协议的文本可以查看RFC2153。

13.8　交 叉 融 合

提示　在本书写作之时，FCoE这个由FCP与以太网结姻所产生的融合协议，正在被一些厂商炒作得沸沸扬扬。FC协议与SCSI协议融合之后形成FCP协议，而FCP协议又与Ethernet融合形成FCoE协议。

如图 13.2 所示是协议融合树。

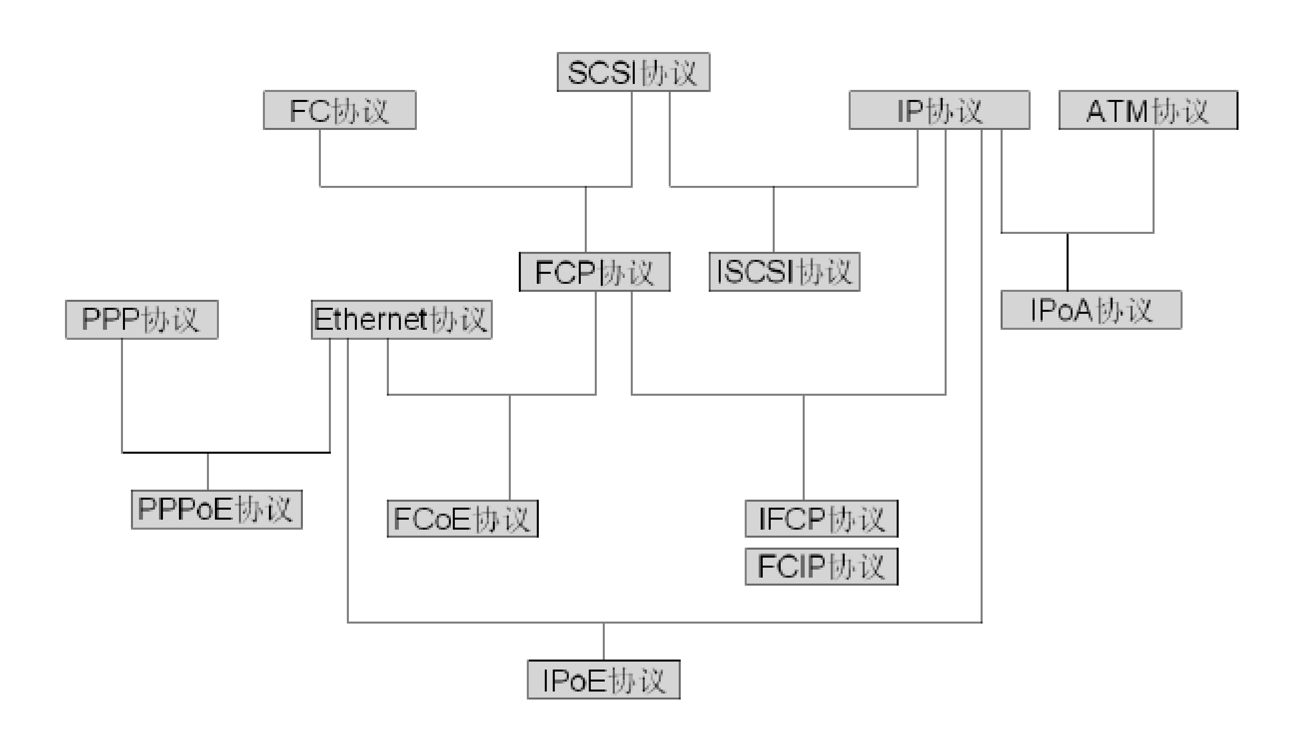


图13.2 协议融合树

从图 13.2 可以看到，FCP 协议与 IP 协议融合的后代是双胞胎。各种协议之间相互融合，甚至产生了交叉，但是一切融合都是为了更好地适合市场需求，因为在资本市场经济时代，只要能满足需求，就能获得利润。“不求最好，但求最适。”



另外，由于 IP 网络的大肆普及，众多的协议动辄就要非 IP 不嫁，而 IP 和以太网绝对是铁哥们儿，所以以太网也借 IP 的光，就凭着自己仅仅 1Gb/s 的带宽到处招摇撞骗，这不，成功地把比它多 3Gb/s 带宽的 FCP 协议给忽悠过来了。不过以太网也在潜心修炼，等练成出关之后，其 10Gb/s 的速率，将会让人望而生畏，但愿那时候以太网统一天下！

13.9 IFCP 和 FCIP 的具体实现

上回说到 IP 根据网络通信协议之间的相互作用，成功地将 FC 协议和 TCP/IP 协议进行了融合，生成一种 FCIP 的 Tunnel 协议和一种 IFCP 的 Map 协议。

蓝图有了，那么具体怎么来将其实现呢？我们知道，不管是 FCIP 的简单 Tunnel 模式，还是 IFCP 的复杂 Map 模式，进行这种 PoP 操作的角色，一定是一端面对 FC 协议的网络，另一端面对 TCP/IP 协议的网络。

1. 协议转换器

同时面对多种协议，并在多种协议之间实现相互融合、相互转换的设备，就称作协议转换器。如果这个转换器只是起到一个桥联的作用，只在一条链路上串联，那么就称其为协议桥接器。如果这种转换器，不但要实现单条链路上的协议转换工作，而且还需要实现一些转发动作，即在多条链路、多个网络之间互相转发数据，则可以称其为协议路由转换器。如果某种协议路由器可以实现多于 2 种协议的网络互联，则称其为多协议路由转换器，因为它能在多种协议之间互相转换并做路由转发。

SAN 要想获得扩展性，即要想将相隔两地很远的两个 SAN 网络通过 IFCP 或者 FCIP 连接起来，就必须在双方的 SAN 系统前端各增加一个协议转换设备，这个设备后端连接各自的 SAN，前端连接 IP 网络，在广域网络上运行 FCIP 或者 IFCP 协议通信，达到协议转换的

目的。

两个独立的系统连接起来，就涉及了两种情况。

- 第一种：两个系统在连接之后，在逻辑上还是独立的，即一个系统不影响另一个系统，但是它们之间可以通过协议转换设备来通信。
- 第二种：两个系统融合成一个大的系统，逻辑上是一体的，只不过相处两地，之间用协议转换设备连接。就像以太网络一样，如果用光缆将两地的两个局域网直接连接起来，两地的系统同在一个广播域中，这样就相当于把两个系统融合起来了。

但是如果两地各自接一个 IP 路由器，广域网链路上承载的是基于广域链路协议之上的 IP 包，那么两地的局域网就没有被融合，可以只是相互通信而已。

提示 有的时候，两地的系统必须融合，而有的时候，不需要融合。是否融合，需要看最终的需求。所以协议转换设备也必须能够处理这两种情况。对于需要融合的情况，协议转换设备不需对两端的 SAN 逻辑做任何附加处理，而只需要将两端的逻辑 Tunnel 或者 Map 到广域网协议上就可以了。而对于不需要融合的情况，协议转换设备就需要对两端系统的逻辑做一系列的处理、屏蔽、虚拟和欺骗了。

2. TCP/IP 和以太网络实例解析

我们不妨拿 TCP/IP 和以太网络来做一个例子。

假如一个公司，在 a 地和 b 地，分别有一个办事处，每个办事处有一台以太网交换机，上面各连接了几台终端。现在为了业务资源共享，公司决定将两地的网络融合起来。公司向 ISP 申请了一条 2Mb/s 的 E1 专线(当然也可以申请 Internet 线路，两端都接入 Internet，然后做 L2VPN 或者 L3VPN)。

公司有两种选择方案。

- 一种是直接用这条专线把两地的交换机连接起来，在这条线路上直接承载以太网帧。
- 另一种选择就是两端各加一个路由器，隔离两边的局域网，但是保持它们之间的通信。

这个公司最终选择了后一种方案，原因就是为了保持双方的独立性，同时保证性能。因为毕竟是两个办事处，如果彻底进行融合，不但不安全，也不利于扩展，而且容易造成广域网流量太大，因为彻底融合之后，以太网广播就要跨广域网来互相传递，这无疑是浪费资源的。在隔离的基础上，同样能够保持双方无障碍的相互通信，只是不能像在一个局域网内那样直接利用 MAC 来点对点通信。如果 a 地某个节点需要和 b 地某个节点通信，a 地的这个节点需要先把数据发给 a 地的路由器，也就是网关设备，然后让网关来转发给 b 地。虽然多增加了一层操作，但是这样做的可扩展性、可管理性都增强了。在路由器上可以做访问控制、地址转换、QOS、策略路由等基于 IP 甚至 TCP 层次的个性化动作。如果是直接局域网融合，则这些特性都不能实现。

3. SAN 系统实例解析

再来看 SAN 的情况。还是这个公司，a 地和 b 地各有一个 SAN 系统。为了实现存储资源直接共享，公司决定将这两个 SAN 联通起来。同样也存在两种情况，即彻底融合或者相对独立的连通。

如果是彻底融合的话，那么广域网链路就完全相当于一条 ISL 链路，只不过通信协议可能是 FCIP 或者 IFCP 协议。

对于 FCIP，任何 FC 帧都将被透明的传递。对于 IFCP，一部分 FC 帧会被屏蔽或者 MAP。但是这些被屏蔽或者 MAP 的帧，都是和底层通信有关的，而上层逻辑性质的帧，IFCP 也需要透传到对端。

这些业务逻辑性质的帧，比如 RSCN 帧，这种用来传递 Fabric 网络中的重要变化信息给已经注册了这项服务的节点；再比如 Plogin，Process Login 等这些都是业务逻辑性质的，和底层通信无关。

彻底融合之后，两个 SAN 系统就融合为了一个系统，那么这个系统就会有一个主交换机，主交换机为系统中其他交换机分配域 ID，并且两个交换机之间需要运行 FSPF 路由协议，不停的发送一些路由控制帧，再加上主交换机选举时产生的帧，主交换机失败时，整个 Fabric 的重建过程中每个交换机发出的各种帧都需要经过广域网链路进行传送。

不但这些帧要占用广域网带宽，而且一旦主交换机发生故障，那么对方的 SAN 系统会进行 Rebuild，包括重新选举主交换机、重新建立路由表等，这个过程中，IO 就会暂时中断。



由于广域网链路速度相对慢，稳定性相对差，所以一旦这条链路发生不稳定的振荡，那么就会造成主交换机重新选举。如果链路频繁闪断的话，那么两端的 SAN 系统根本无法正常工作了。

所以说，两地 SAN 系统彻底融合的话，一旦某地的系统故障，就会影响到另一个系统的正常运行，而且要占用额外多的宝贵的广域网资源。由于访问存储资源对性能和延迟要求较高，所以彻底融合两个 SAN，最好只在局域网内进行，交换机间的链路最好是裸光缆或者高速链路，否则最好采用另外一种融合方式，即逻辑独立、全局连通的融合方式。

13.10 局部隔离/全局共享的存储网络

将 SAN 系统彻底融合，扩展性差、管理性差，而且耗费广域网链路资源。所以这个公司同样也选择了相对独立的连通方式。下面来看一下，相对独立的融合，到底是个什么概念，它的作用机制是怎样的。

"a 地的 SAN 交换机(E 端口)－a 地协议转换器－广域网链路－b 地协议转换器－(E 端口)b 地 SAN 交换机"这种拓扑不管是彻底融合，还是独立融合都一样，只不过协议转换器在两种方式下所作的工作不一样。彻底融合方案中，协议转换只 Tunnel 或者 Map 通信底层的协议逻辑，而不管上层业务逻辑，也就是只要从 E 端口收到了帧，协转就将其 Tunnel 或者 Map 到 IP 协议中发送给对端。而相对独立的融合，不但要 Tunnel 或者 Map 底层协议逻辑帧，它还要理解 FC 的上层逻辑，做到"报喜不报忧"。

独立融合/全局共享

所谓独立融合，就是说两端的 SAN 系统都可以独立运作，而不依靠另一方，或者受另一方的影响。这样就不能像彻底融合那样一端为主交换机，一端为非主交换机，而要让两端独立起来。由于两端的 Fabric 中都各自只有一台 SAN 交换机，所以两端的 SAN 交换机都是主交换机，各自为政。

既然这样，怎么能和对方的 SAN 进行通信呢？协议路由器自有其招数。协议路由器与 SAN 交换机之间通过 E 端口连接，它欺骗两端 SAN 交换机，让交换机认为它正在连接着另一台交换机，而这个由协议转换器虚拟出来的交换机级别比它低，所以它自己认为自己就是主交换机。虚拟交换机和 SAN 交换机之间运行 FSPF 路由协议，所以这个虚拟交换机就获得了 SAN 交换机下面所有连接的终端节点信息。

获得这些信息之后，a 地的协转通过广域网链路将这些信息通告给 b 地的协议转换器。b 地的协议转换器同样和 b 地的 SAN 交换机之间运行着 FSPF 路由协议，同样也欺骗了 b 地交换机。b 地协议转换器收到了 a 地协议转换器发来的关于 a 地 SAN 交换机上所连接的所有节点信息之后，就利用和 b 的 SAN 交换机之间的 FSPF 路由协议，将这些节点信息通告给 b 地 SAN 交换机，所以 b 交换机就有了 a 交换机上节点的信息，同样 a 交换机也会拥有 b 交换机上节点的信息，这样，a 和 b 交换机之间就可以通信了，其实它们都不知道中途有两个中介在骗它们。

如果其中一个 SAN 系统发生故障，那么这个系统中的协转设备，会将这个重大消息屏蔽，不告诉对端的 SAN 系统。因为一旦被对方系统得知，便会发生 Fabric 的重建过程，影响本端 SAN 系统的 IO。有了 SAN 路由器，远端 SAN 访问的超时，并不会影响本地 SAN 的访问。此即所谓的“报喜不报忧”。同样，一个 SAN 系统中的诸如 RSCN 等广播类的帧，也会被协转设备根据策略而终结在本地，不会跨越广域网链路通告给对方。协转设备还应该具有访问控制功能。

这种方案被称作“SAN 路由”，因为它具有像 IP 路由类似的功能和架构。

13.11　多协议混杂的存储网络

如图 13.3 所示，其中的中枢引擎是两个互相连接的多协议路由器。这个多协议处理机，就像一台计算机的 CPU，Fabric 和以太网络就像计算机的 IO 总线，磁盘便是计算机的外设和输入设备，各种存储控制器便可以理解为计算机上的各种 IO 控制器，前端的 Fabric 和以太网便是前端的 IO 总线，主机服务器则是输出设备。即磁盘上的数据，经过输入总线输入 CPU 进行运算，然后通过输出总线，输出给主机服务器。这又是一个轮回，不折不扣的轮回，循环嵌套，永无止境。

图 13.3 所示的拓扑，可以说是一个大的统一的拓扑。存储网络不外乎就是图 13.3 中列出的元素。磁盘经过一层层的 IN 和 OUT，一层层的虚拟化或者桥接透传，最终被主机看作是一个 LUN 或者目录。不妨将其抽象，隐去复杂的部分，就形成了图 13.4 的拓扑。

再抽象一下，如图 13.5 所示。

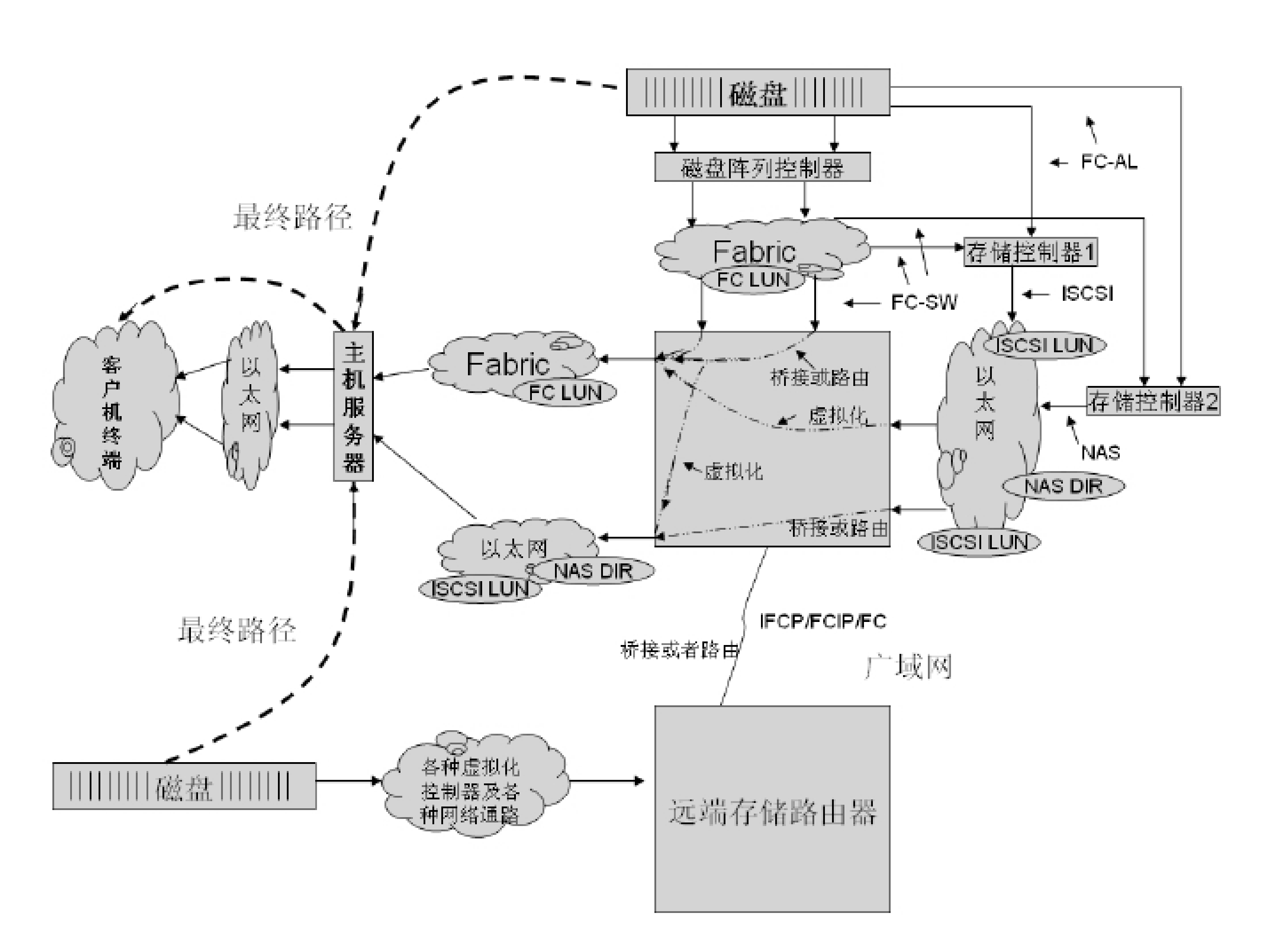


图 13.3 多协议混杂的存储网络

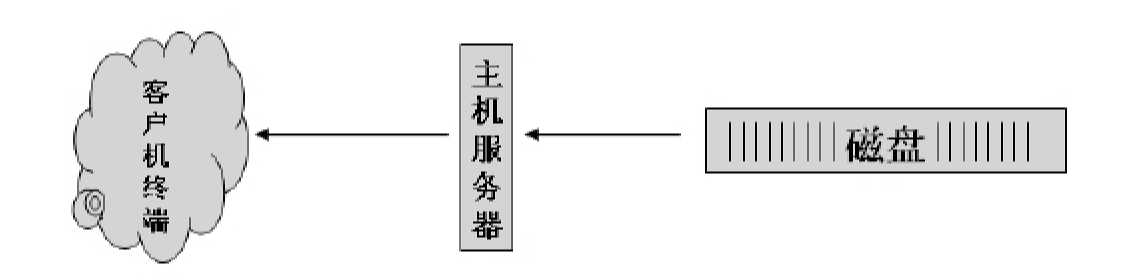


图 13.4 一次抽象后的系统架构

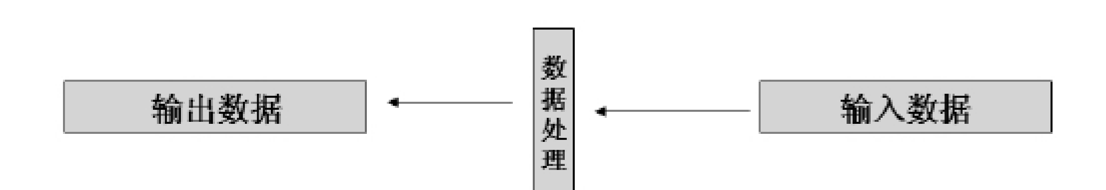
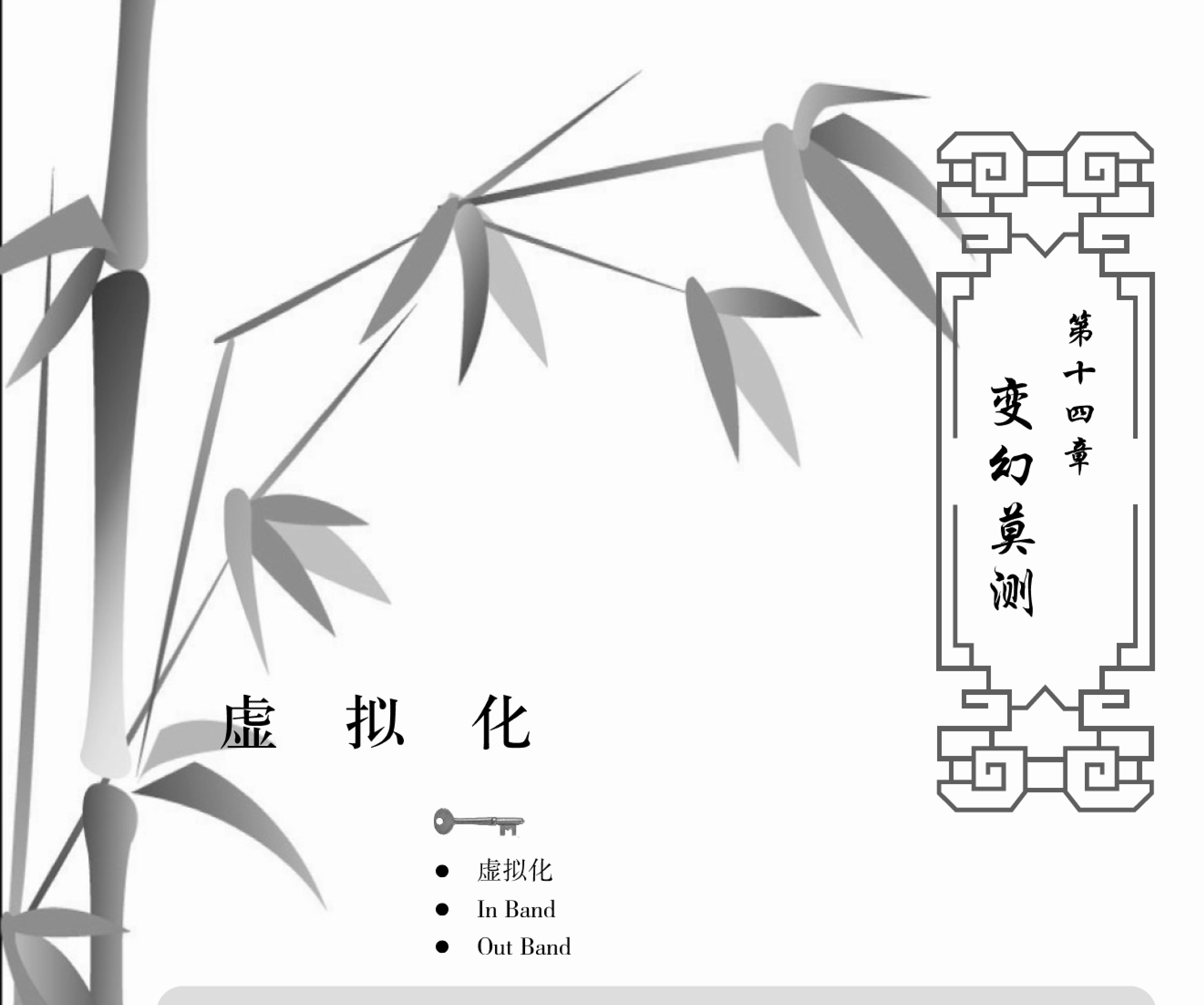


图 13.5 本质模型

读 书 笔 记



虚　拟　化

- 虚拟化
- In Band
- Out Band

“计算机科学中的任何问题，都可以通过加上一层逻辑来解决。”

——计算机科学家 David Wheeler

目前形形色色的软件层出不穷，可是它们都脱离不了一个基础，那就是计算机硬件系统。如 CPU、内存和各种 IO 接口，以及连接它们，使它们之间相互通信的总线。

CPU 内部是大量的集成逻辑电路，CPU 不断受到一种电信号的“刺激”，这种刺激经过 CPU 内部的逻辑电路的一层层的传递转换，最终输出另一种电信号。这种输入、输出动作，是有一定逻辑的。通过编写汇编代码，可以实现对 CPU 内部逻辑电路的刺激，并引起一系列的逻辑输出。将这些逻辑映射到人们所能理解的知识上，比如输出 1 代表对，输出 0 代表错误，或者如果输出 1 则继续刺激，输出 0 则停止刺激等，这样就构成了从基本的逻辑电路，到复杂的思维逻辑的映射，由简单逻辑的层层嵌套，构成了复杂逻辑。将汇编语言，用人类容易理解的语言抽象出来，就形成了高级语言。将高级语言的意思，转换成低级语言的过程，就是编译。比如说：冬瓜，而用低级语言表示冬瓜这个意思，就是：“撇，横折，捺，点，点，撇……”。

14.1　操作系统对硬件的虚拟化

我们知道，早期的计算机系统，其实是没有操作系统的，因为操作系统本身也是靠计算机硬件执行的一种程序。操作系统就是一种可以提供给其他程序方便编写并运行的程序。由程序来运行程序，而不是由程序自己来运行，这是操作系统提供的一种虚拟化表现。

1. 早期计算机单任务模式

对于早期计算机来说，只能允许执行一个任务，整个计算机只能被这个程序独占。比如开机，从软盘或者其他介质上执行程序，直到执行完毕或者人为中断。执行完后拿出介质，再插入另一张介质，重新载入执行另一个新的程序。在执行程序的过程中，一旦意外终止，就要重新运行。

如果有 10 个人要用一台计算机来执行程序，第一个人拿着他的软盘，上面有一个数学题计算程序，他插入软盘，然后重启机器，机器从软盘特定的扇区载入程序代码执行，结果显示在显示器上，比如这个程序 2 个小时运行完毕，第一个人从显示器上抄下结果，走了。后面有 9 个人在排队等待用计算机。然后第二个人同样拿着他的软盘，插入软驱，重启……每次更换程序，都需要重新启动机器，简直就是梦魇。再者，如果某个程序运行期间，会有空闲状态，则其他程序也仍然需要等待，CPU 只能在那里空振荡。

2. 操作系统的多任务模式

操作系统的出现解决了这两个问题。操作系统本身也是一段程序，计算机加电之后，首先运行操作系统，随时可以载入其他程序执行，也就是说它可以随时从软盘上读取其他程序的代码，并切换到这段代码上让 CPU 执行，执行完毕后则立即切换回操作系统本身。但是每次也总是要等待这个程序执行完毕，才能接着载入下一个程序执行。当被载入的程序执行的时候，不能做任何其他的事情，包括操作系统本身的程序模块，任何产生中断的事件，都会中断正在运行的程序。

程序执行完毕之后，会将 CPU 使用权归还操作系统，从而继续操作系统本身的运行。这种操作系统称为单任务操作系统，典型代表就是 DOS。

一旦在 DOS 中载入一个程序执行，如果没有任何中断事件发生，则这个程序就独占 CPU，执行完毕之后，回到 DOS 操作系统，接着可以继续执行另外一个程序。经过这样的解决，执行多个程序，期间就再也不用重新启动机器了。

在这个基础上，操作系统又将多个程序一个接一个的排列起来，成批的执行，中途省掉了人为载入程序的过程，这个叫做批处理。批处理操作系统，相对于单任务操作系统来说，可以顺序的、无须人工干预的批量执行程序，比简单的单任务操作系统又进了一步，但是其本质还是单任务性，即一段时间之内，仍然只会观察到一个应用程序在运行，仍然只是一个程序独占资源。

再后来，操作系统针对系统时钟中断，开发了专门的中断服务程序，也就是多任务操作系统中的调度程序。时钟中断到来的时候，CPU 根据中断向量表的内容，指向调度程序所在的内存地址入口，执行调度程序的代码，调度程序所作的就是将 CPU 的执行跳转到各个应用程序所在的内存地址入口。每次中断，调度程序以一定的优先级，指向不同程序的

入口。这样就能做到极细粒度的应用程序入口切换，如果遇到某个程序还没有执行完毕就被切出了，则操作系统会自动将这个程序的运行状态保存起来，待下次轮到的时候，提取出来继续执行。比如每 10ms 中断一次，那么也就是说每个应用程序，可以运行 10ms 的时间，然后 CPU 运行下一个程序，这样依次轮回。微观上，每个程序运行的时候，还是独占 CPU，但是这个独占的时间非常小，通常 10ms，那么一秒就可以在宏观上“同时”运行 100 个程序。这就是多任务操作系统。多任务操作系统的关键，就是具有多任务调度程序。

通过这样的虚拟化，运行在操作系统之上的所有程序都会认为自己是独占一台计算机的硬件运行。

3. 虚拟化的好处

上面说了计算机硬件以及操作系统，其实计算机系统从诞生的那一天开始，就在不断地进行着虚拟化过程，时至今日，计算机虚拟化进程依然在飞快发展着。

硬件逻辑被虚拟化成汇编语句，汇编语句再次被封装，虚拟化成高级语言的语句。高级语言的语句，再次被封装，形成一个特定目的的程序，或者称为函数，然后这些函数，再通过互相调用，生成更复杂的函数，再将这些函数组合起来，就形成了最终的应用程序。程序再被操作系统虚拟成一个可执行文件。其实这个文件代表了什么呢？到了底层，其实就是一次一次的对 CPU 的电路信号刺激。也就是说，硬件电路逻辑，一层层的被虚拟化，最终虚拟成一个程序。程序就是对底层电路上下文逻辑的另一种表达形式。

虚拟化的好处显而易见，虚拟化将下层的复杂逻辑，转变为上层的简单逻辑，方便人类读懂，也就是说“科技，以人为本”。任何技术，都是为了将上层逻辑变得更加简单，而不是越变越复杂，当然使上层越简单，下层就要做更多的工作，就越复杂。

整个计算机技术，从开始到现在，就是一个不断的抽象、封装、虚拟、映射的过程，一直到现在还在不断抽象封装着，比如 Java 等比 C 抽象封装度更高的高级语言，当然使用起来也比 C 方便和简单多了，但是随之而来的，其下层就要复杂一些，所以 Java 代码一般运行速度慢，耗费资源也大，但是对于现在飞速发展的硬件能力，是不成问题的。

同样，CPU 也不仅仅只是一味地增加晶体管数量这么简单。CPU 制造者也在想尽办法将一些功能封装到 CPU 的逻辑电路中，从而出现了更多的指令集，这些指令集就像程序函数一样，不必理解它内部到底怎么实现的，只需要发给 CPU，CPU 就会启动逻辑电路计算。到目前为止，Intel 的 CPU 已经发展到了酷睿 2 代四核。主频 1.6GHz 的酷睿双核 CPU，竟然性能比主频 3GHz 甚至超频到 4GHz 的奔腾 4 代 CPU 还高。所以 CPU 的设计除了提高主频之外，更重要的是内部逻辑的优化，集成度的提高，更多的抽象和封装。



虚拟化的思想在计算机的各个方面都是存在的，比如经典的 OSI 模型，就是一个不折不扣的抽象虚拟模型。尤其是 TCP/IP 协议给上层抽象出来的 Socket 接口，即“插座接口”，也可以理解为，只要将插头接上这个插座，就会和网络接通，而不必管它是怎么实现的，就像将交流电插头插入市电插座一样，插上就获得了电压，而不用管市电电网的拓扑，更不用关心国家总电网的拓扑了。

14.2 计算机存储子系统的虚拟化

上面说了很多关于汇编和操作系统的虚拟抽象，下面说说计算机系统中的存储子系统中的虚拟化。

存储子系统的元素包括：磁盘、磁盘控制器、存储网络、磁盘阵列、卷管理层、目录虚拟层、文件系统虚拟层。我们就从下到上，一一描述这几个元素，看看存储子系统是怎么抽象虚拟的。

1. 磁盘控制器

磁盘控制器的工作就是根据驱动程序发来的磁盘读写信息，向磁盘发送 SCSI 指令和数据。这个部件看似没有什么可抽象虚拟的东西，其实磁盘控制器完全可以对其驱动程序隐藏其下挂的物理磁盘，而虚拟出一个或者多个虚拟磁盘。由控制器来完成虚拟磁盘和物理磁盘的映射和抽象虚拟。RAID 就是一个点型代表，控制器将物理磁盘组成 RAID Group，然后在 RG 的基础上，虚拟出多个 LUN，通告给主机驱动。

2. 存储网络

早期的存储子系统，没有网络化，而目前的存储系统，网络化已经非常彻底。从磁盘到磁盘阵列控制器，从磁盘阵控制器到主机总线适配器，都已经嵌入了网络化元素。比如使用 FC 协议，或者 TCP/IP 协议、SAS 协议、Infiniband 协议等。那么在这一层上，有什么可以抽象的么？网络化只是为部件之间提供了一种可扩展的传输通路而已，貌似在这个层面上不能做出什么大文章来。

实则不然，这一层也是有所深究的。在交换式 SAN 中，不管是基于 TCP/IP 协议的还是基于 FC 协议的 SAN，网络中的任何节点，都是通过交换设备来互相通信，这是节点间通信的必经之路。如果在交换设备上做点手脚，就完全可以达到虚拟化的效果。

要抽象一种逻辑，那么一定要理解这种逻辑，所以我们可以在 FC 交换机或者以太网交换机上，嵌入 SCSI 协议感知模块。比如某个 N 节点向另一个 N 节点 Report LUN 的时候，交换机收到这个 Frame，则可以感知这个 N 节点的 LUN 信息。如果此时网络中还有另一个节点的 LUN 信息，则可以在交换机这一层，达到这两个节点的 LUN 的镜像。也就是说，SCSI 发起设备向目标设备传输的数据，经由交换机的时候，交换机内嵌的虚拟化模块，会主动复制对应的帧到另一个节点的 LUN 上，让这两个 LUN 形成镜像，当其中一个节点故障的时候，交换机因为知道此时还有一个备份镜像 LUN 存在，所以并不会向发起者通告失败，而是默默的将发起者的数据重定向到这个镜像的 LUN，发起设备并不会感知，这样，就达到了基于网络层的虚拟化抽象。

当然，网络层的虚拟化，并不只是镜像，比如将某些 N 节点的 LUN 合并成一个池，然后动态的从这个池中再划分出虚拟 LUN，向发起者报告等。基于这些思想，已经开发出了智能 FC 交换机。

3. 磁盘阵列

磁盘阵列可以说本身就是一个小计算机系统(JBOD 除外)，这个系统五脏俱全，是对存储子系统的抽象虚拟化最佳的表现。磁盘阵列，简要地说，就是将大量磁盘进行组织管理，抽象虚拟，最终形成虚拟的逻辑磁盘，最后通过和主机适配器通信，将这些逻辑磁盘呈现给主机。这个功能和前面提到的磁盘控制器的功能类似，但是磁盘阵列能比狭义的磁盘控制器提供更多的特色功能，况且简单的插在主机 IO 总线上的那种 RAID 磁盘控制器，其接入磁盘数量有限，功能也有限。

大型磁盘阵列，有自己的控制器，有的利用嵌入式技术，将特别定制的操作系统及其核心管理软件嵌入芯片中，来管理整个控制器并实现其功能；有的则干脆利用现成的主机来充当盘阵控制器的角色，比如 IBM 的 DS8000 系列盘阵，内部就是用的两台 IBM P 系列小型机作为其组织管理磁盘的控制器，其上运行 AIX 操作系统和相应的存储管理软件。

不管是嵌入式，还是主机式的，盘阵控制器所担任的角色都是类似的。这个中心控制器，不直接参与连接每块磁盘，而是利用后端适配器来管理下挂的磁盘，由后端适配器向其上级汇报。

这些适配器，就是由中心控制器驱动的二级磁盘控制器，这些磁盘控制器作为中心 CPU 的 IO 适配器，直接控制和管理物理磁盘，然后由中心控制器统一实现 RAID、卷等高级功能(有些盘阵则可以将简单的 RAID 功能直接下放给二级控制器来做)。后端适配器与中心控制器 CPU 之间通过某种总线技术连接，比如 PCIX、PCIE 总线等。中心控制器对这些磁盘进行虚拟抽象之后，通过前端的接口，向最终使用它的主机进行通告。中心控制器不但可以实现最基本的 RAID 功能，而且可以实现很多高级功能，比如 LUN 镜像、快照、远程复制、CDP 数据保护、LUN 再分配等。在磁盘阵列上实现虚拟化，是目前最广泛的一种存储系统虚拟化形式。

有些产品甚至学成了借花献佛的本领。比如 NetApp 公司 V 系列 NAS 网关、HDS 公司的某些存储设备以及 IBM 公司的 SVC。这些设备既可以当主机使用，又可以当存储设备使用，面对存储，它就是主机，而面对主机的时候，它就是存储，如图 14.1 所示。

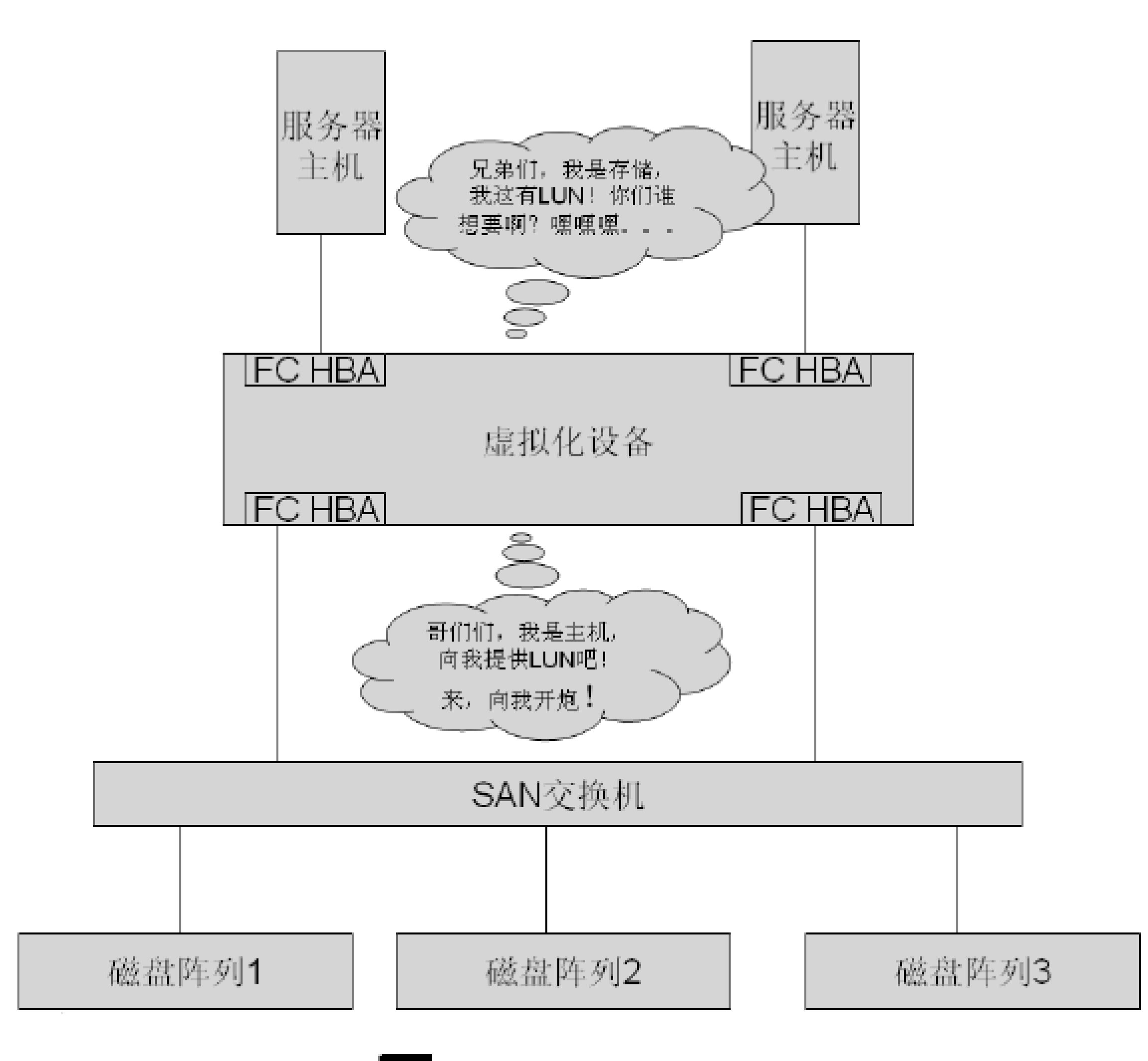


图14.1 借花献佛

提示：乍一感觉这台虚拟化设备也真够无赖的，明明自己没有磁盘却能踩着别人的脚向外提供 LUN，明明就是自己向别人租赁来的然后又装修了一把，转租出去。但是我们非常需要这种设备，因为它帮了大忙。

假如，图 14.1 中磁盘阵列 1 的容量为 1TB，磁盘阵列 2 的容量为 1TB，而某台主机需要一个容量为 2TB 大小的 LUN，这怎么办呢？我们可以在主机上安装卷管理软件，让 VM 把这两个 1TB 的 LUN 合并成一个 2TB 的卷即可。但是这么做需要耗费主机资源，且虚拟好的新 LUN 只能给这台主机使用。而一些旧的低端设备，由于其容量和性能等已经不能满足要求，如果可以将这些设备挂到这台虚拟化设备上，作为一个二线存储资源，这样就将所有的存储资源整合到了一起，统一管理和分配。

要想获得足够的性能和灵活性，就需要我们图 14.1 所示的虚拟化设备了。

这个设备在盘阵端(后端)的 FC HBA 卡处于 FC Initiaor 模式，即在后端，这台设备以主机模式出现。而在前端(主机端)，这台设备的 FC HBA 卡为 FC Target 模式，即它以盘阵的角色出现。这样，这台设备就可以从其后端掌管 LUN，然后将这些 LUN 合并并再次灵活分割，呈交给其前端的多台主机使用。除了简单的合并再分割 LUN 之外，这台设备还可以做许多其他数据管理操作，比如将两个 LUN 镜像、快照、CDP 等。

4. 卷管理层

卷管理层是指运行在应用主机上的功能模块。它负责底层物理磁盘或者 LUN 的收集和再分配。经过盘阵控制器虚拟化之后生成的 LUN 提交给主机使用，主机可以对这些 LUN 进行再次抽象和虚拟，也就是重复虚拟化，比如对其中两个 LUN 进行镜像处理，或者对其中的多个 LUN，做成一个软 RAID 系统。再或者将所有 LUN 合并，形成一个大的资源池，然后像掰面团一样掰成多个卷，这个过程和磁盘控制器。盘阵控制器所做的虚拟化动作类似，但是这个动作是在主机上实现的。典型的卷管理软件有 LVM，或者第三方的软件，比如 Veritas 公司的 VxVM。

5. 文件系统

数据只是存储到磁盘上就完了么？显然不是。打个比方，有位记者早晨出去采访，手中拿了一摞纸，他每看到一件事就记录下来。对于“怎么将字写在纸上”这个问题，他是这么解决的，他用笔在格子上写字，写满一行再写下一行，还不够就换一张纸。对于“怎么让自己在纸上写字”这个问题，是他自己通过大脑(控制器)，通过神经网络(SCSI 线缆)，操纵自己的手指(磁头臂)，拿着笔(磁头)，看见有格子，就向里写。这两个问题都解决了。可是这一天下来，他回去想看看一天都发生了什么，他拿出记录纸，却发现，信息都是零散的，根本无法阅读，有时候读到一半，就断了，显然当时是因为格子不够用了，写到其他地方了，造成了信息记录的不连续，有的地方还有删除线，证明这一块作废了，那么有效的记录到底在哪里呢？记者方寸大乱，数据虽然都完好的记录在纸上，但是他们都是不连续的、凌乱的，当时是都记下来了，但是事后想要读取时却没辙了。

磁盘记录也一样，只解决磁盘怎么记录数据和怎么让磁盘记录数据，是远远不够的，还应该考虑“怎么组织磁盘上的数据”。

还是用这个记者的例子来说明。我们都能想到，将凌乱的记录组织成完整的一个记录，

只需要在相应的地方做一下标记，比如“此文章下一段位于某某页，第几行”，就像路标一样，一次一次的指引你最终找出这个完整的数据，这个思想称为“链表”。

如果将这个链表单独的做成一个记录，存放到固定位置，每次只要参考这个表，就能找出一条数据在磁盘上的完整分布情况。利用这种思想做出来的文件系统，比如 FAT 文件系统，它把每个完整的数据称为文件。文件可以在磁盘上不连续的存放，由单独的数据结构来描述这个文件在磁盘上的分布，这个数据结构就是文件分配表。File Allocate Table，也就是 FAT 的由来。或者用另一种思想来组织不连续的数据，比如 NTFS，它是直接给出了一个文件在磁盘上的具体扇区，开始—结束，开始—结束，用这样的结构来描述文件的分布情况。

文件系统将磁盘抽象成了文件柜，同一份文件可能存放在一个柜子的不同抽屉中，利用一份特别的文件来记录“文件－对应抽屉”的分布情况，这些用来描述其他文件分布情况及其属性的文件，称为元文件(Metadata)。元文件一般情况下要存放在磁盘的固定位置，而不能将其分散，因为最终要有一个绝对参考系统。但是有些文件系统，甚至将元文件也可以像普通文件一样，在磁盘上不连续的分布。前面还说过一定要有一个绝对参考系统，也就是固定的入口，所以这些特殊的文件系统，其实最上层还是有一个绝对参考点的，这个参考点将生成元文件/在磁盘上的分布情况记录，从而定位元文件，再根据元文件，定位数据文件，这样一层一层的嵌套，最终形成文件系统。

最终一句话，文件系统是对磁盘块的虚拟、抽象、组织和管理。用户只要访问一个个的“文件”，就等于访问了磁盘扇区。而访问文件，这个动作是非常容易理解的，也是很简单的，用户不必了解这个文件最终在磁盘上是存放到哪里，怎么存放的，怎么访问磁盘来存放这个文件，这些统统都是由文件系统和磁盘控制器驱动程序来做。

6. 目录虚拟层

不管是 Windows 系统、UNIX 系统，还是 Linux 系统，其内部都有一个虚拟的目录结构。在 Linux 中叫做 VFS，即 Virtual File System。

虚拟文件系统，顾名思义也就是说这个文件系统目录并不是真实的，而是虚拟的。任何实际文件系统，都可以挂载到这个目录下，真实 FS 中的真实目录，被挂载到这个虚拟目录下之后，就成为了这个虚拟目录的子目录。这样做的好处是增强灵活性。其次，操作系统目前处理外部设备，一般都将其虚拟成一个虚拟文件的方式，比如一个卷，在 Linux 中就是/dev/hda 这种文件。对这个文件进行读写，就等于直接对设备进行了读写。

存储子系统的虚拟化，可以在“磁盘－盘阵控制器－存储网络－主机总线适配器－卷管理层－文件系统层－虚拟目录层和最终应用层”各个环节的虚拟抽象工作，使得最终应用软件，只要通过文件系统访问文件，就可以做到访问最底层的磁盘一样的效果。有时候还可以重复虚拟化。

14.3 带内虚拟化和带外虚拟化

- 所谓带内即 In band，是指控制信令和数据走的是同一条路线。所谓控制信令，就是说用来控制实际数据流向和行为的数据。典型的控制信令，比如 IP 网络中的各

种 IP 路由协议所产生的数据包，它们利用实际数据线路进行传输，从而达到各个设备之间的路由统一，这就是带内的概念。

- 带外即 Out band，是指控制信令和实际数据走的不是同一条路，控制信令走单独的通路，受到“优待”。

带内和带外，只是一种叫法而已，在电话信令中，带内和带外是用“共路”和“随路”这两个词来描述的。共路信令指的是控制信令和实际数据走相同的线路；随路信令则指二者走不同的线路，信令单独走一条线路。随路又可以称作“旁路”，因为它是单独一条路。

明白了上面这些概念，用户就可以理解所谓“带内虚拟化”和“带外虚拟化”的概念了。

- 带内虚拟化，就是说进行虚拟化动作的设备，是直接横插在发起者和目标路径之间的，斩断了二者之间的通路，执行中介操作，发起者看不到目标，而只能看到经过虚拟化的虚拟目标。所以在带内虚拟化方式下，数据流一定会经过路径上的所有设备，即所有设备是串联在同一条路径上的，虚拟化设备插入这条路径中，作为一个“泵”，经过它这的时候就被虚拟化了。
- 带外虚拟化，则是在这个路径旁另起一条路径，也就是所谓的旁路。用这条路径来走控制信号，而实际数据还是由发起者直接走向目标。但是这些数据流是受控制信令所控制的，也就是发起者必须先“咨询”旁路上的虚拟化设备，经过“提示”或者“授权”之后，才可以根据虚拟化设备发来的“指示”直接向目标请求数据。带外虚拟化方式中，数据通路和信令通路是并联的。

带内虚拟化的例子非常多，目前的虚拟化引擎几乎都是带内虚拟化。IBM 的 SVC(San Volume Controller)、Netapp 的 V-series、HDS 公司的 USP 系列等，它们都是带内虚拟化引擎。

1. 带外虚拟化系统 SanFS

带外虚拟化的一个典型的例子，是 IBM 公司的 SanFS 系统。如图 14.2 显示了 SanFS 的基本架构。

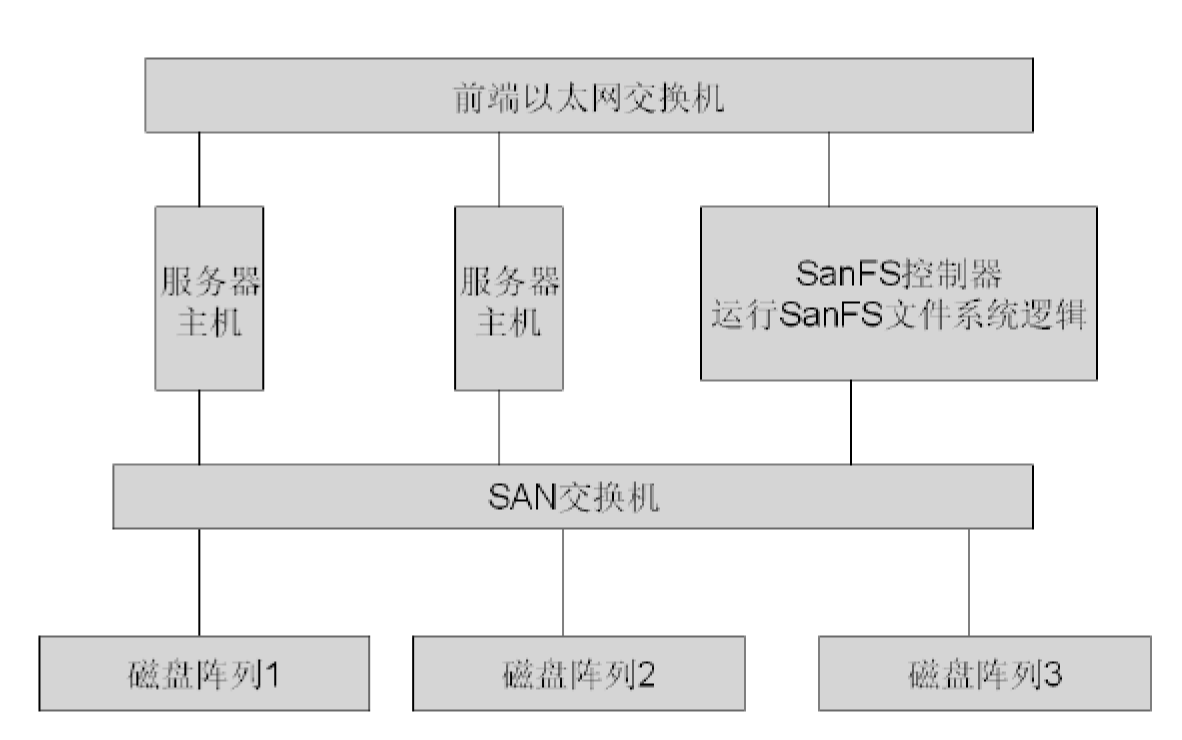


图14.2　IBM 公司的 SanFS 架构示意图

SanFS 其实根本没有什么高深的地方，说白了，SanFS 就是一个网络上的文件系统，也就是说，常规的文件系统都是运行在主机服务器上的，而 SanFS 将它搬出到了网络上，用一台专门的设备来处理文件系统逻辑。

然而，这个“网络上的文件系统”却绝对不是“网络文件系统”。网络文件系统是典型的带内虚拟化方式，因为网络文件系统对上层屏蔽了底层卷，只给上层提供一个目录访问接口，上层看不到网络文件系统底层的卷，只能看到目录。而 SanFS 架构中，上层既能看到文件系统目录，又能看到底层卷(LUN)，如图 14.3 所示。

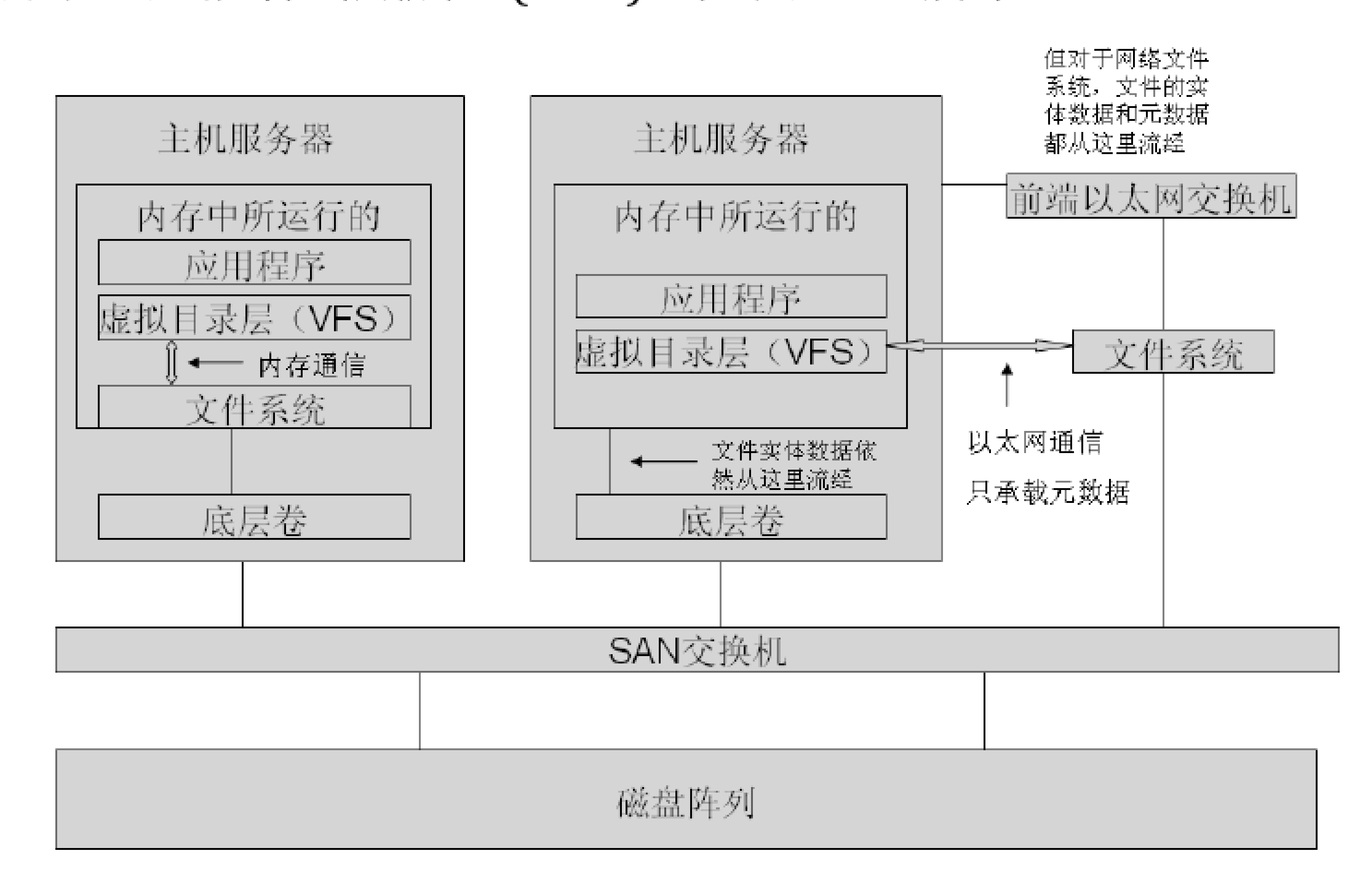


图14.3 SanFS 架构示意图

图 14.3 中，左边的服务器是一台普通的服务器，右边是一台使用 SanFS 文件系统的服务器。右边的服务器上的文件系统已经被搬到了外面，即运行在一台 SanFS 控制器上，这个控制器与服务器都接入一台以太网交换机。当虚拟目录层需要与文件系统层通信的时候，通信路径不再是内存，而是以太网了。由于文件系统已经被搬出主机，所以任何与文件系统的通信都要被重定向到外部，并且需要用特定的格式，将请求通过以太网发送到 SanFS 控制器，以及从控制器接受相应的回应，所以在使用 SanFS 的服务器主机上，必须安装 SanFS 管理软件(或者叫做 SanFS 代理)。

下面用几个实例来说明 SanFS 是如何作用的。

实例 1

服务器运行 Window 2003 操作系统，使用 SanFS 作为文件系统的卷的盘符为“S”盘。在 Windows 中双击盘符 S，此时 VFS 虚拟目录层便会发起与 SanFS 控制器的通信，因为需要获取盘符 S 根目录下的文件和目录列表。所以 VFS 调用 SanFS 代理程序，通过以太网络向 SanFS 控制器发送请求，请求 S 根目录的文件列表，SanFS 控制器收到请求之后，将列表通过以太网发送给 SanFS 代理，代理再传递给 VFS，随即就可以在窗口中看到文件和目录列表了。

实例 2

某时刻，某应用程序要向 S 盘根目录下写入一个大小为 1MB 的文件。VFS 收到这个请求之后，立即向 SanFS 控制器发送请求，SanFS 控制器收到请求之后，计算出应该使用卷上的哪些空闲块，将这些空闲块的 LBA 号码列表以及一些其他必要信息通过以太网传送到服务器。服务器上的 SanFS 收到这些信息后，便调用操作系统相关模块，将应用的数据从服务器的内存中直接向下写入对应卷的 LBA 地址上。

SanFS 系统是一个典型的带外虚拟化系统，服务器主机虽然可以看到底层卷，但是管理

这个卷的文件系统，却没有运行在主机上，而是运行在主机之外。主机与这个文件系统之间通过前端以太网通信，收到文件系统的指示之后，主机才按照指示将数据直接写入卷。

SanFS究竟有何意义呢？SanFS是不是有点多此一举呢？放着主机内存这么好的风水宝地不用，却自己跑出去单独运行，和别人通信还得忍受以太网的低速度(相对内存来说)，这是何苦呢？煞费苦心的SanFS当然有自己的算盘，这么做的原因有三个。

- 将文件系统逻辑从主机中剥离出来，降低主机的负担。
- 既然将文件系统从主机剥离出来，为何不干脆做成NAS呢？NAS同样也是将文件系统搬移出主机。答案是因为向NAS传输数据，走的是以太网，速度相对FC SAN要慢。所以SanFS的设计是只有元数据的数据流走以太网，实际数据依然由主机自行通过SAN网络写入盘阵等存储设备。这样就加快了数据传输速度，比NAS有优势。
- 其实SanFS一个最大的特点，就是支持多台主机共享同一个卷，即同一时刻可以由多台主机共同读写(注意是读写)同一个SanFS卷。这也是SanFS最大的卖点。共享同一个LUN的所有主机，都与SanFS控制器通信以获得访问权限，所以SanFS干脆就自己单独占用一台专用设备，放在网络上，也就是SanFS控制器，这样可以让所有主机方便地与它连接。

2. SanFS与NAS的异同

SanFS与网络文件系统究竟有何不同呢，如图14.4所示。

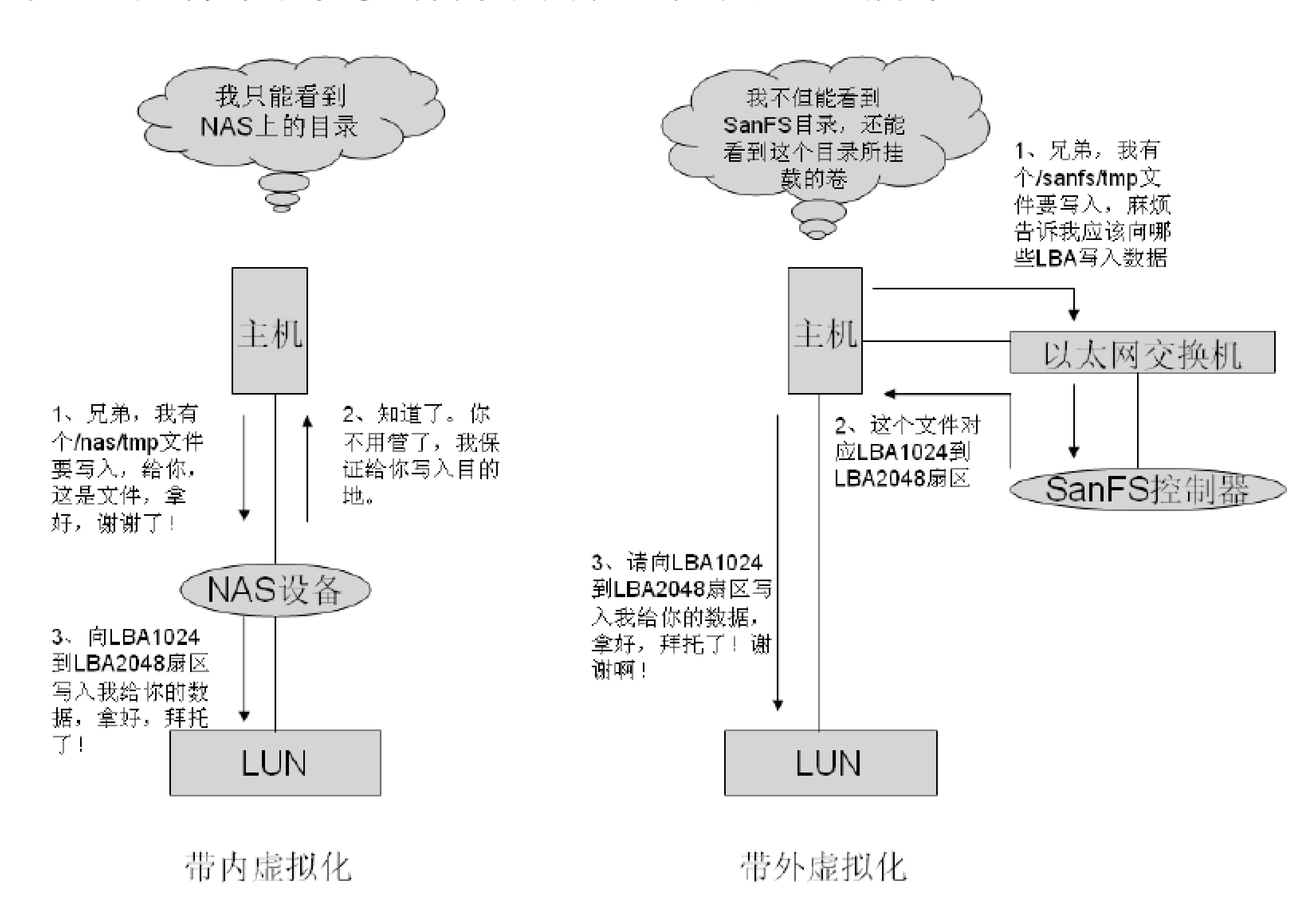


图14.4 SanFS与NAS的异同

显然，NAS(网络文件系统)与SanFS是截然不同的。主机向NAS写入数据，其实要经历两次写的过程，第一次写是主机将数据通过以太网发送给NAS的时候，第二次写是NAS将收到的数据写入自己的硬盘(本地磁盘或者SAN上的LUN卷)。而SanFS只写入一次数据，而且是通过FC SAN，而不是相对慢速的以太网。

3. 轮回和嵌套虚拟化

以太网在传统上是用来承载 IP 的，但是有一些技术是将以太网承载到 IP 之上的，比如 VPLS。VPLS 属于一种极端变态的协议杂交方式。它嵌套了多次，也轮回了多次。

VPLS 可以说是对以太网 VLAN 技术的一种上层扩展。传统 VLAN 使用 VLAN 标签来区分不同的域，VPLS 则可以直接通过用不同的 IP 来封装以太网头来区分各个以太网域。

这个技术也从一个侧面表明了 TCP/IP 在当今网络通信领域所不可动摇的绝对地位。

14.4 硬网络与软网络

1. 硬件网络设备

所谓的硬件网络设备，其功能终究还是靠软件来实现的。很多网络硬件设备，尤其是路由设备，本质上就是一台 PC 或者 PC Server。其上运行着专门处理网络数据包的程序。就这样，若干底层网络设备互相连接，组成了整个基础网络，也就是硬件网络环境。

在硬件网络环境的基础上，若干 PC 接入硬件网络，实现相互通信。也就是说，用一部分 PC 充当网络硬件设备，其他 PC 利用这些充当网络设备的 PC 实现通信。这就是一种嵌套的表现，也就是“网中有网”。

2. 软件网络程序

Message Queue 和 Message Broker 在硬件网络设备的基础上，模拟出一个纯软件的网络转发引擎。这就是一种轮回的表现。

- MQ(Message Queue)是一种消息转发软件引擎。这个引擎运行在主机操作系统之上。其功能就是充当一个消息转发器。客户端通过 TCP/IP 与这个转发器相连，将消息传送到这个转发器上，然后转发器根据策略，将消息转发到其他客户端上。这种消息转发器，也就类似于网络交换机。只不过 MQ 的链路层由 TCP/IP 来充当。
- MB(Message Broker)是一种应用逻辑转发引擎。这个引擎虽然也是用来转发消息的，但是它不仅仅是底层转发，还能做到应用层次的转发。这类似于邮件服务器，只不过它可以转发各种格式和方式的数据包。

14.5 用多台独立的计算机模拟成一台虚拟计算机

1. HPC 环境

点组成线，线组成面，面组成体，体与体之间组成网，然后就是进化。同样，HPC 环境也是这种模式。在一个典型的 HPC 环境中，包含众多的计算机，这些计算机各有分工，总体来说，HPC 环境中的计算机可以分为两大类，一种是专门用来计算数据的，为 CPU 密集运算；另一种是专门用于存储计算过程中，所需要提取或者存放的数据的，为 IO 密集运算。前者称为计算节点，后者称为存储节点。而为了最大利用硬件资源，有些 HPC 环境中会存储节点，也兼用来做计算节点。

可以将一个 HPC 环境中的所有计算节点，看作一台大的虚拟计算机的 CPU 和内存，而

将所有存储节点看作虚拟计算机的硬盘。虚拟计算机的 CPU 和内存(计算节点)，通过某种连接链路向虚拟计算机的硬盘(存储节点)读写数据，从而计算出结果。对于一台单独的物理计算机来说，CPU 内存与存储设备之间的连接为高速 IO 总线，比如 PCIE。但是对于由多台独立节点组成的 HPC 系统来说，虚拟 CPU 与虚拟存储设备之间的连线就不可能是内部 IO 总线了，而是一种外部的高速网络传输方式。有些 HPC 利用 Infiniband 网络作为计算节点与存储节点之间的连接方式，有些则干脆使用以太网。前者一般用于 IO 密集型的运算；后者一般用于 CPU 密集型运算，也就是说，运算过程中需要读写的数据不多。

2. 典型的 Web+APP+DB 架构

这种架构是一种典型的 IT 架构。客户端通过 Web 服务器获取一个图形化显示网页，应用逻辑由 APP(Application)服务器处理，并将结果通过 Web 服务器显示到客户端的网页上，APP 服务器需要的数据则通过访问数据库服务器来获得。

也可以将 Web 服务器看作一台显示终端，将 APP 服务器看作 CPU 和内存，将 DB 服务器看作硬盘。这样，一个由 Web+APP+DB 服务器所组成的虚拟计算机便诞生了。

14.6 用一台独立的计算机模拟出多台虚拟计算机

1. Vmware 虚拟机软件

Vmware 通过模拟一套硬件系统，将程序对这个硬件系统 CPU 发送的指令经过一定的处理之后，并加以虚拟都传到物理 CPU 上执行。利用这种方式，可以在一台物理计算机上虚拟出多个虚拟机。

目前 Windows Server 2008 操作系统已经自带了 HyperV 虚拟化引擎。类似 Vmware 的 ESX。目前很多操作系统都集成了 Native 的虚拟化引擎。

2. 世界本身就是一个轮回嵌套的虚拟化系统

不但在计算机领域中有虚拟化，在其他学科中同样有虚拟化。化学领域中，科学家把观察到的现象和计算出来的公式，虚拟化成原子和分子。

总之，一切都是虚拟化的结果，我们观察到的世界其实就是我们利用基本数学公式虚拟出来的。人们首先在大脑中演绎出数学，然后虚拟化出了物理学，然后再用数学和物理学，虚拟化出化学等其他各种学科。这就像用汇编语言来抽象数字电路逻辑，再用高级语言来抽象汇编语言，然后将现实中的逻辑用计算机高级语言表达出来，让计算机来模拟出现实逻辑。

14.7 用磁盘阵列来虚拟磁带库

VTL，即 Virtual Tape Library，虚拟磁带库。传统的物理磁带库为全机械操作，比如机械手、驱动器、磁带等。其速度相对磁盘来说要慢很多，如果需要备份的数据量非常大，而备份窗口又很小，那么只能通过提高磁带库的速度来解决。但是要提高磁带库的速度，只能同时用多个驱动器同时操作，需要成本高，不方便。虚拟磁带库的出现为的就是解决

上述这些问题。VTL 使用磁盘来存储数据而不是磁带，并虚拟出机械手、磁带驱动器、磁带这三样在物理上都不存在的东西。在备份软件等使用磁带库的应用程序，不会发现物理设备到底是盘阵还是真实的磁带库。而虚拟化之后，前端的程序接口不变，后端的速度和灵活性却大大增加了。

图 14.5 是一台物理磁带库的正视图。图 14.6 显示了仓门打开后其内部构件示意图，可以看到一根竖直的柱子，这个柱子就是机械手的滑轨。机械手可以沿着柱子上下滑动并且可以左右转动，以抓取右侧磁带槽中的磁带。在图 14.8 以及第 16 章的图 16.7 中可以看到另一种设计的机械手和驱动器。图 14.7 所示是物理带库的两个驱动器和电源后视图。



图14.5　一台物理磁带库的正视图

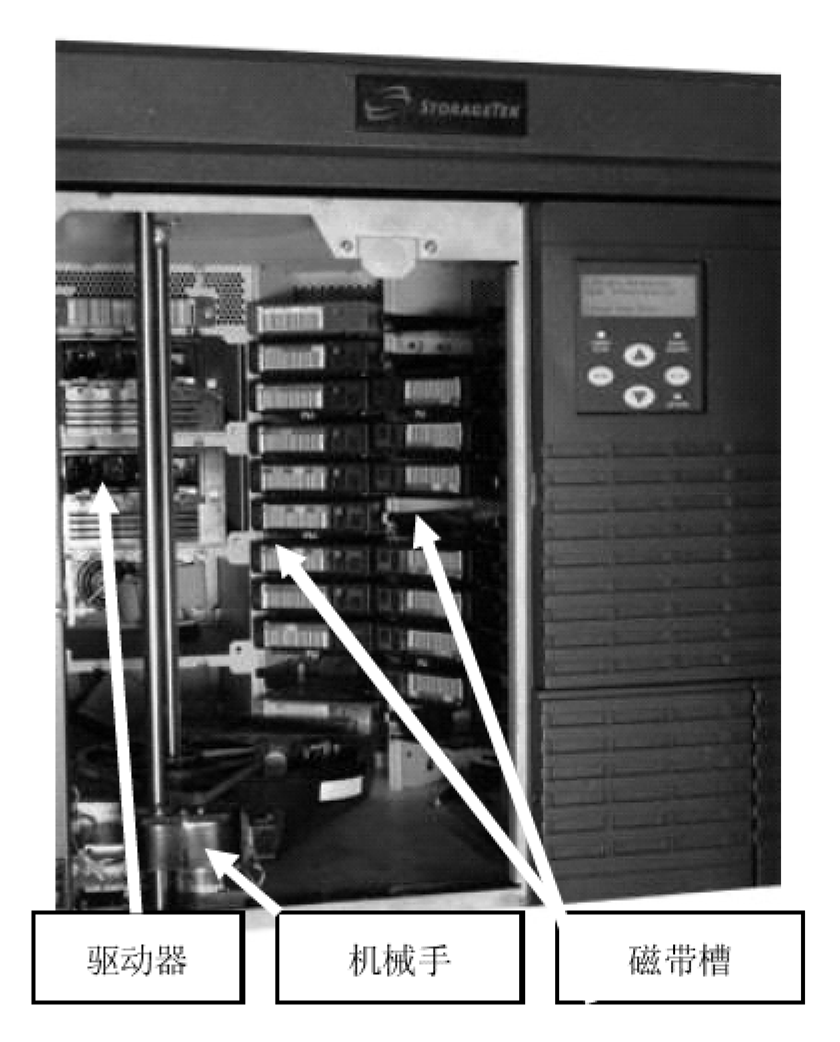


图14.6　物理磁带库的内视图



图14.7　物理磁带库后视图(驱动器和电源)



图14.8　另一种设计方式的磁带库机械手

14.7.1 NetApp VTL700 配置使用实例

比如有 NetApp VTL700 系列虚拟磁带库一台，机头连接了两个扩展柜。共 28 块 500GB 的 SATA 磁盘。

1. 第一步：创建 RAID 组，为虚拟磁带创建底层存储空间

磁盘阵列必须做 RAID。这是任何情况下都要保证的。RAID 不仅仅可以提高速度，更重要的是为了保护数据，因为任何一块硬盘损坏，如果没有 RAID，都会造成数据丢失。VTL

使用的磁盘阵列也不例外。

1】 VTL700 可以利用 Web 界面来管理，管理主界面如图 14.9 所示。

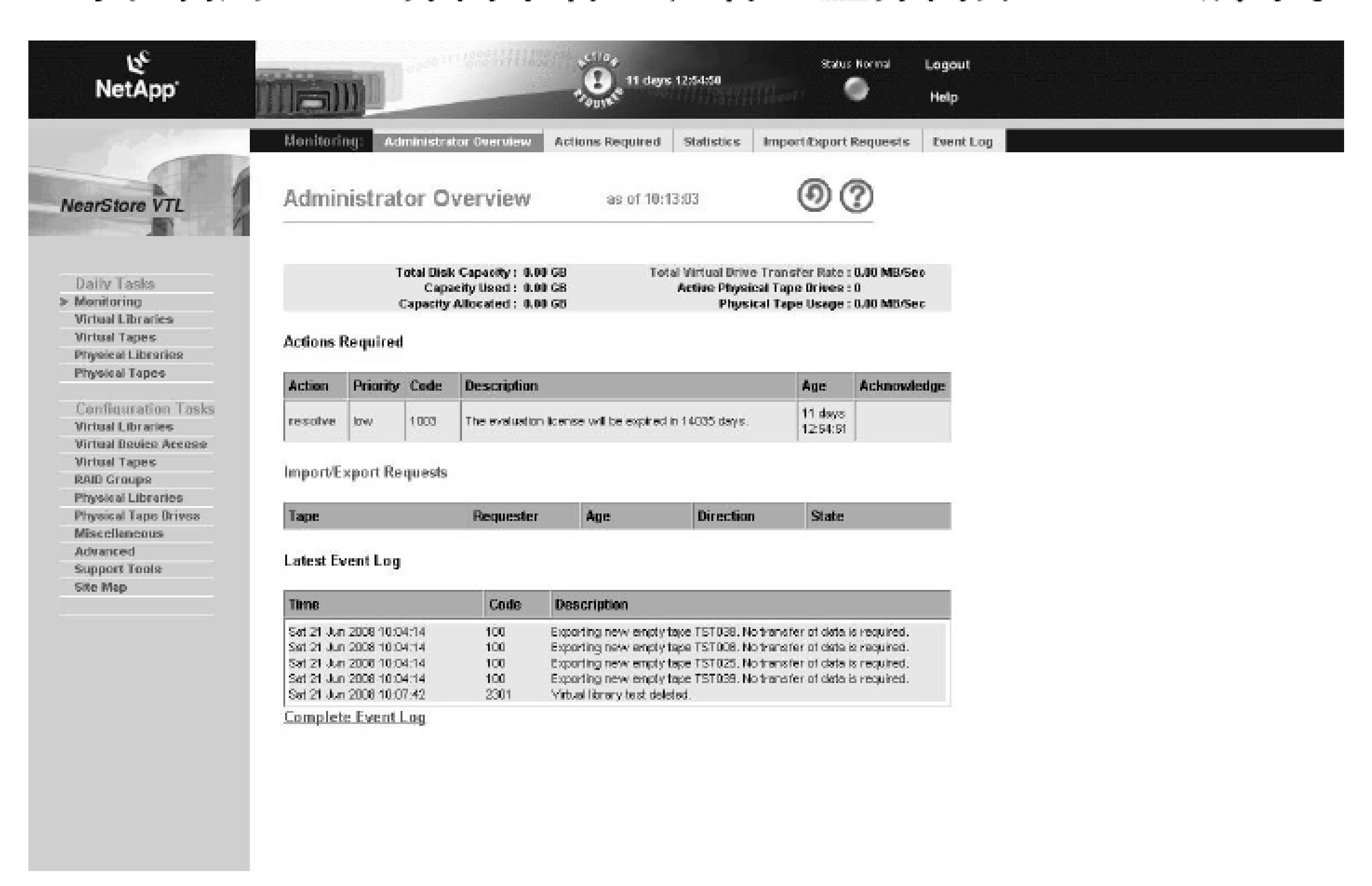


图14.9　VTL700 管理主界面

2】 查看当前系统的虚拟磁带容量等信息，如图 14.10 所示。由于还没有配置完成，所以图 14.10 中没有给出任何虚拟之后的容量等信息(RAID 组和虚拟磁带总容量都为 0.00GB)。

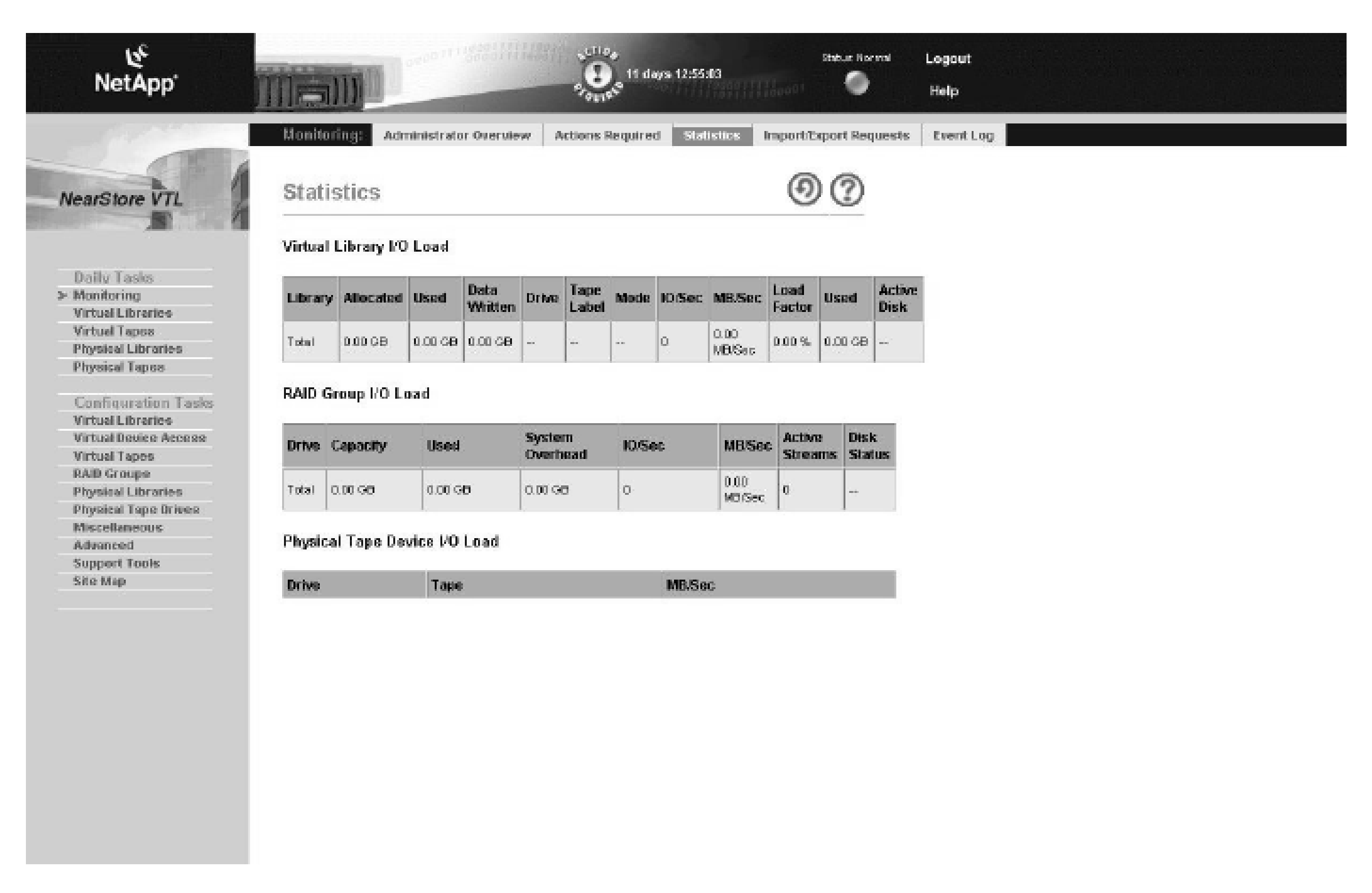


图14.10　系统配置容量信息

3】 从页面左侧栏中选择 RAID Groups 标签，右侧会显示出当前系统中所配置的 RAID 组，如图 14.11 所示。由于当前还没有配置任何 RAID 组，所以两台扩展柜中的所有磁盘均显示为灰色(空闲状态)，蓝色的磁盘为 Spare 盘。

4】 单击图 14.11 上方的 Create Raid Group，出现如图 14.12 所示的页面。

5】 单击图 14.12 中的 Automatically Create RAID Group 按钮，让系统自动创建 RAID 组。出现如图 14.13 所示的窗口，提示用户创建 RAID 组的规则。

6】 单击 Apply 按钮出现如图 14.14 所示的页面，可以看到系统已经创建好了一系列的 RAID 组，且每个 RAID 组用不同颜色表示。

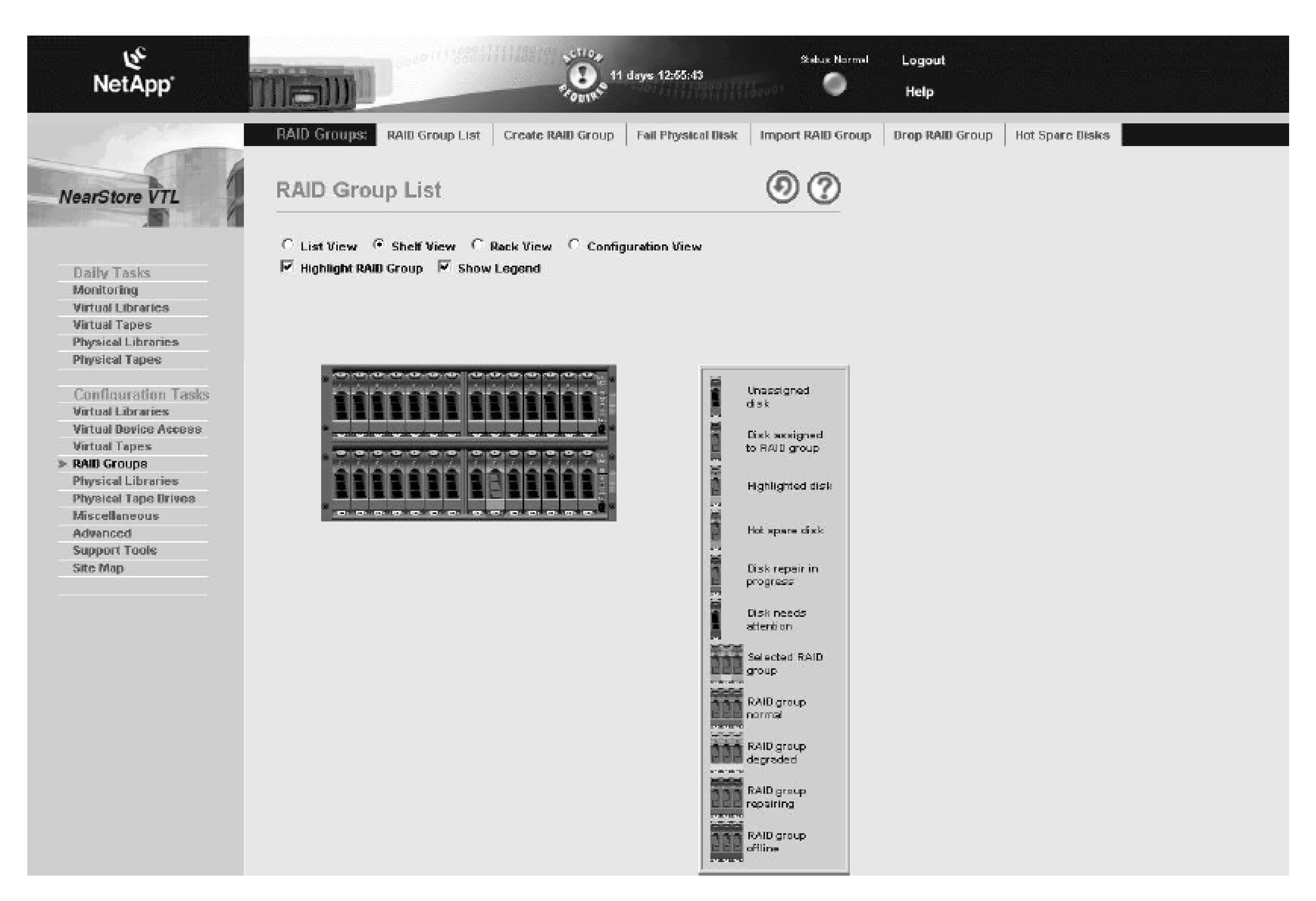


图14.11　系统当前 RAID 组信息

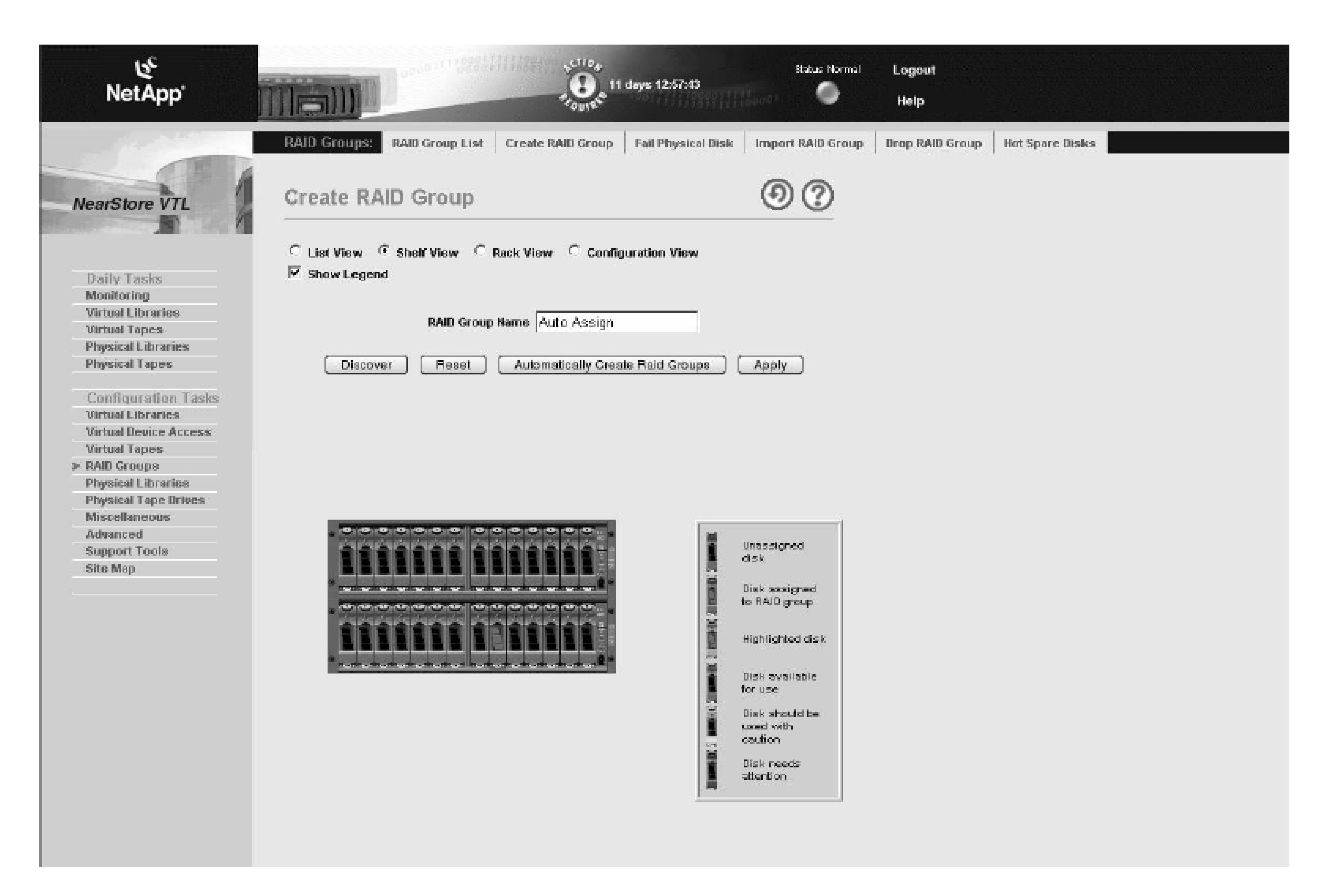


图14.12　创建 RAID 组页面

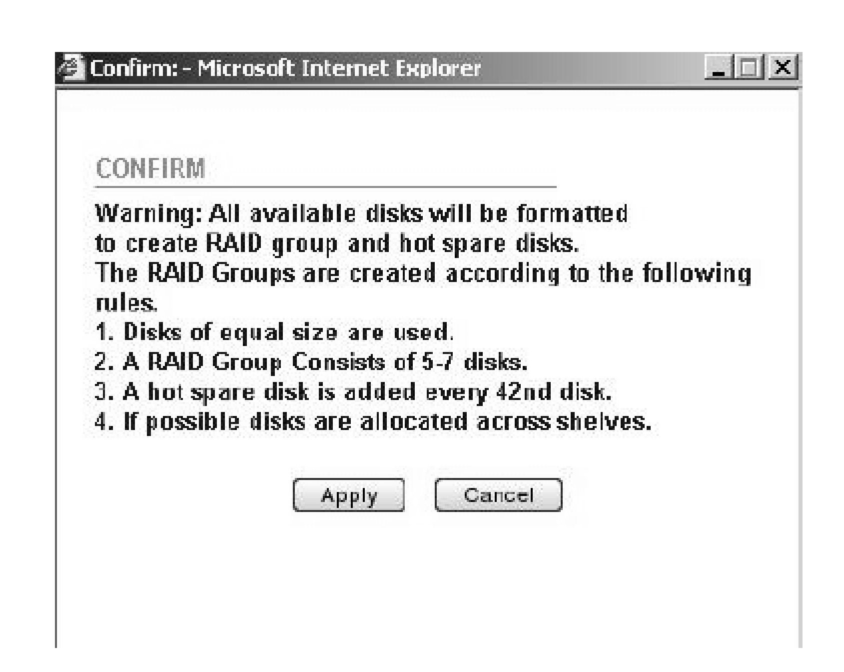


图14.13　创建 RAID 组规则

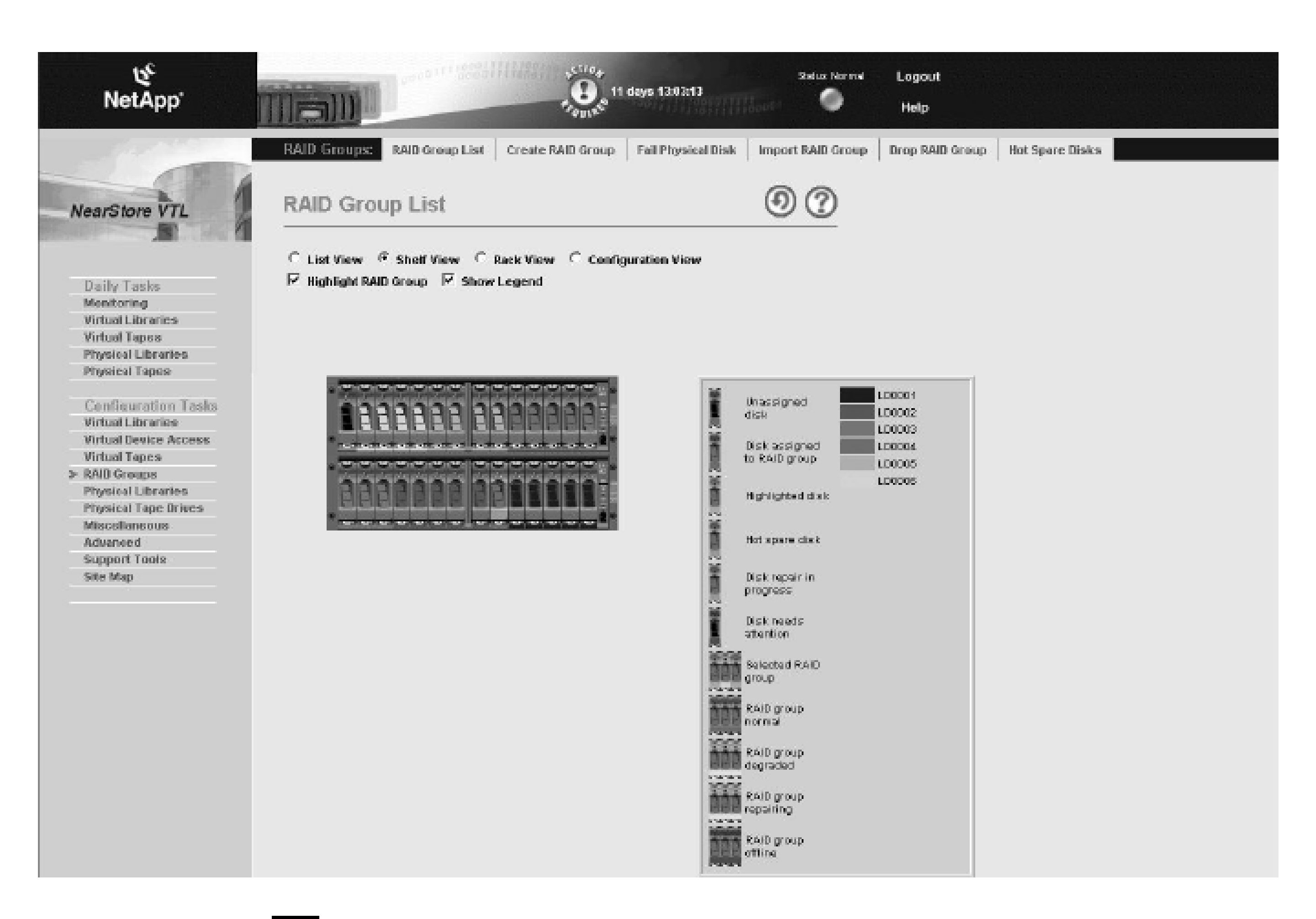


图14.14 系统自动创建的 6 个 RAID 组

7】 单击 Drop RAID Group 标签，进入删除 RAID 组页面，可以删除已经创建好的 RAID 组，如图 14.15 所示。



图14.15 删除 RAID 组页面

8】 单击 Drop，删除对应的 RAID 组。删除之后也可以再次手动或者自动创建 RAID 组。图 14.16 所示为手动创建的 7 个 RAID 组。

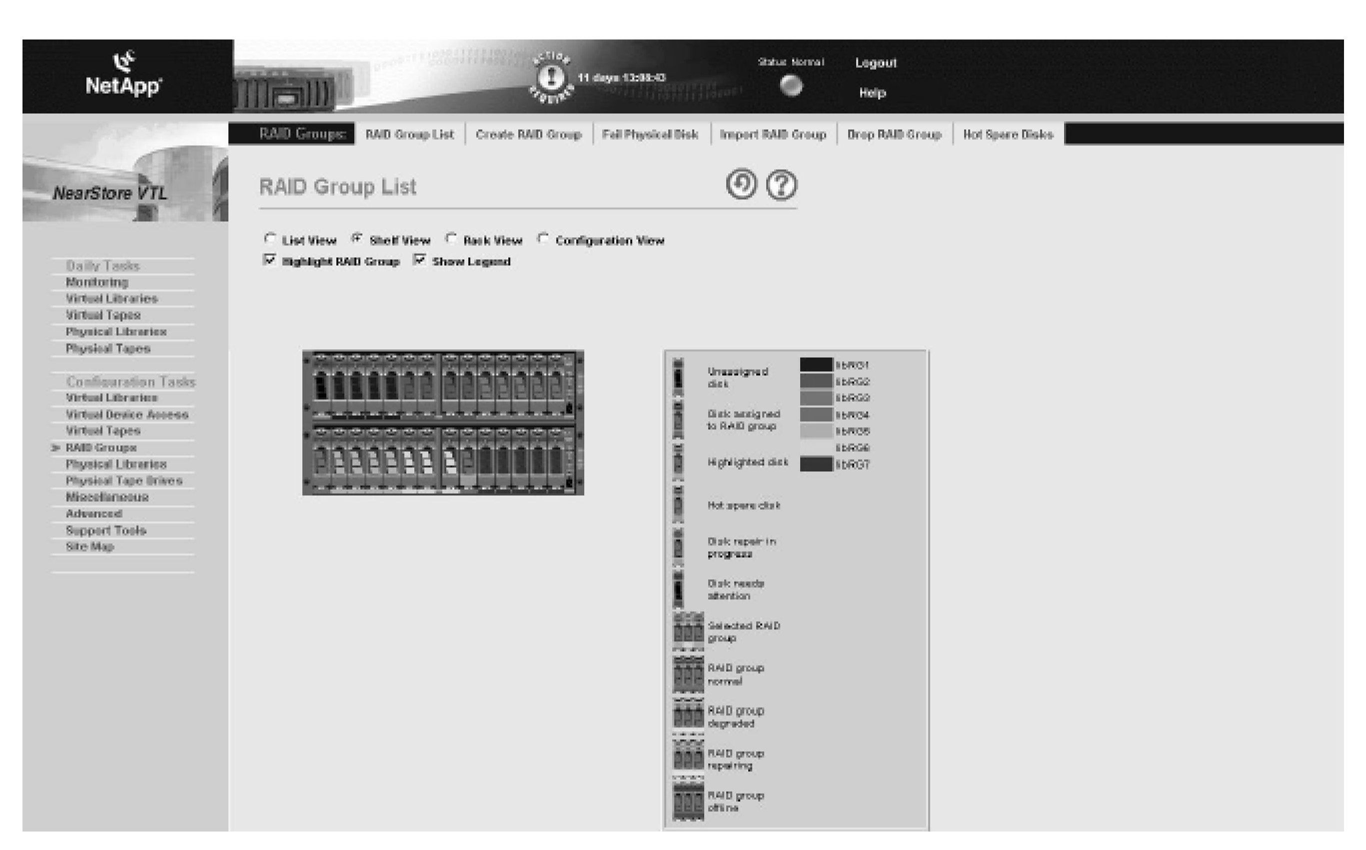


图14.16 手动创建的 7 个 RAID 组

9】 回到 Monitor 页面，查看系统当前的配置容量信息，RAID Group 栏目中已经显示出了系统当前的可用磁盘容量，如图 14.17 所示。

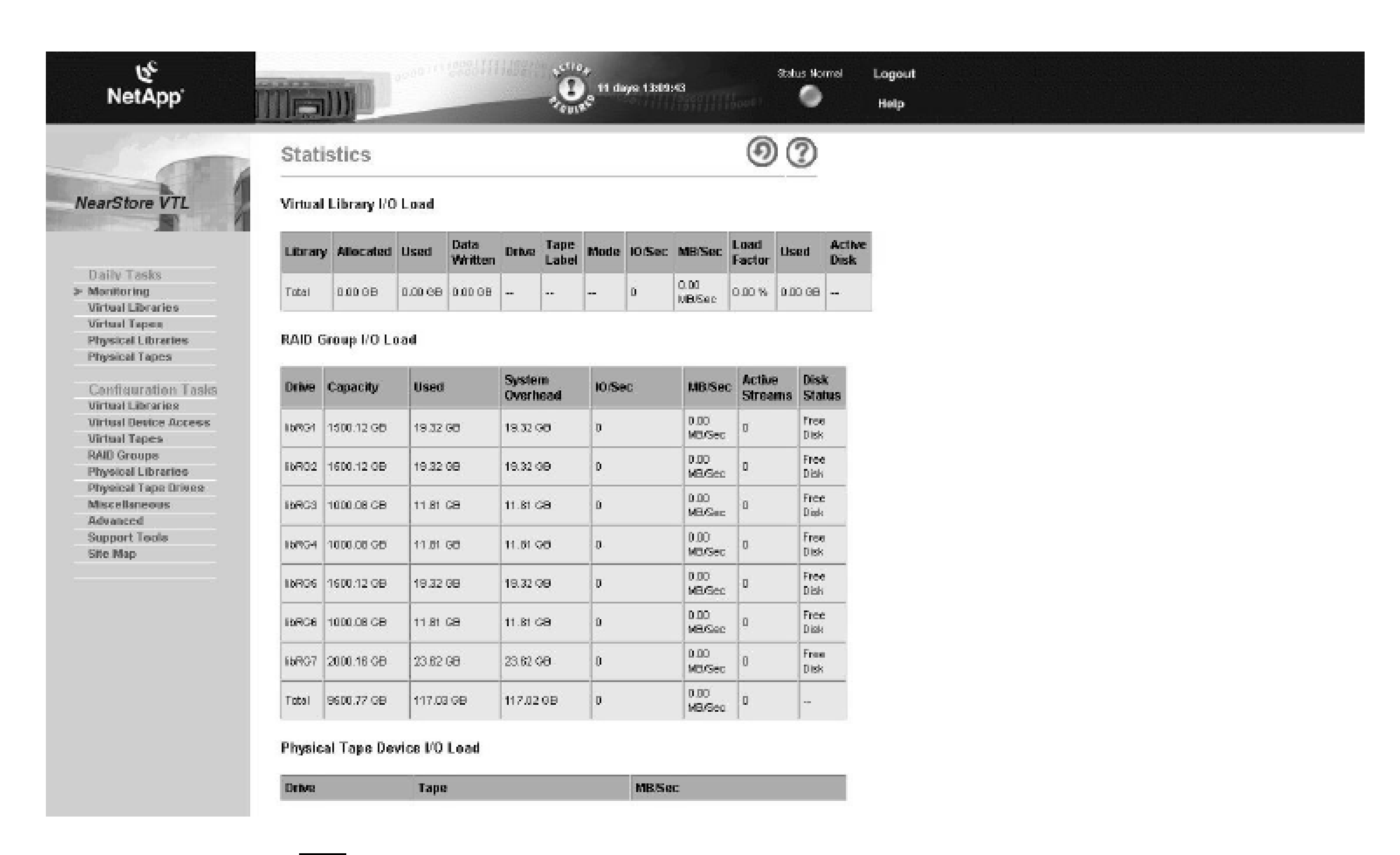


图14.17　RAID 组配置完后的系统容量信息

2. 第二步：创建虚拟磁带库和虚拟磁带

创建完 RAID 组之后，就为系统提供了基本的存储空间。在这个存储空间之上，还需要创建一台或者多台虚拟带库，然后创建虚拟磁带。

1】 创建虚拟磁带库。如图 14.18 所示创建一台名为 lib1 的虚拟带库，带库的厂家信息为 NetApp。虚拟磁带驱动器类型为 IBM LTO1，这个类型必须对应一款物理磁带驱动器，下拉列表框中有多种支持的物理驱动器可选。Slot Count 即磁带槽位，这里定为 8。Drive Count 即驱动器数量，这里定为 2。Assigned to Port 表示将这个虚拟磁带库在哪个前端端口“提交”出去，即主机端可以从 VTL 上的哪个前端端口识别到这台虚拟磁带库。Fully Loaded with Virtual Tapes 表示是否自动创建对应的虚拟磁带并插满磁带槽(磁带槽和磁带都是虚拟的)。Start Tape Label 表示起始标签名称。物理磁带库中，每盘磁带都有各自的标签，机械手通过这些标签来识别每一盘磁带，虽然 VTL 不需要用标签来识别每个虚拟磁带，但是备份软件等程序需要知道这些标签，所以 VTL 还是要给每个虚拟磁带分配标签，这里用“L1”来起始，这样，第二盘虚拟磁带系统就会自动为其分配“L2”的标签，依此类推。



图14.18　创建虚拟磁带库

2】 单击 Apply 按钮之后，一台虚拟磁带库就创建完并可以使用了。此外，还可以创建更多的虚拟磁带，如图 14.19 所示。进入左侧栏 Virtual Tapes 标签，右侧页中可以选择新磁带归属于哪个虚拟带库。这里选择刚刚创建的“lib1”，然后起始标签设为“M1”，数量为 8 盘，单击 Apply 按钮。这样就向“lib1”这台虚拟带库中增加了 8 盘磁带，加上原有的一共 16 盘。



图14.19　增加虚拟磁带

3】 单击左侧栏中的 Virtual Libraries 标签，右侧会出现一张虚拟磁带库拓扑图。如图 14.20 所示，右侧机柜中有 8 个磁带槽和 2 个驱动器，与创建带库时的选项一一对应。左边的推车上还有 8 盘磁带，这是刚才追加的 8 盘磁带，而虚拟带库中的磁带槽放不开，所以就放到了虚拟推车上。推车上方是一个存储箱，可以用鼠标把任意一盘磁带拖动到推车、磁带槽或者驱动器中。如果拖动到存储箱中，则这盘虚拟磁带便被删除了(如图 14.21 所示)。

4】 返回到 Monitor 页面中可以看到当前系统中的虚拟带库信息，如图 14.22 所示。

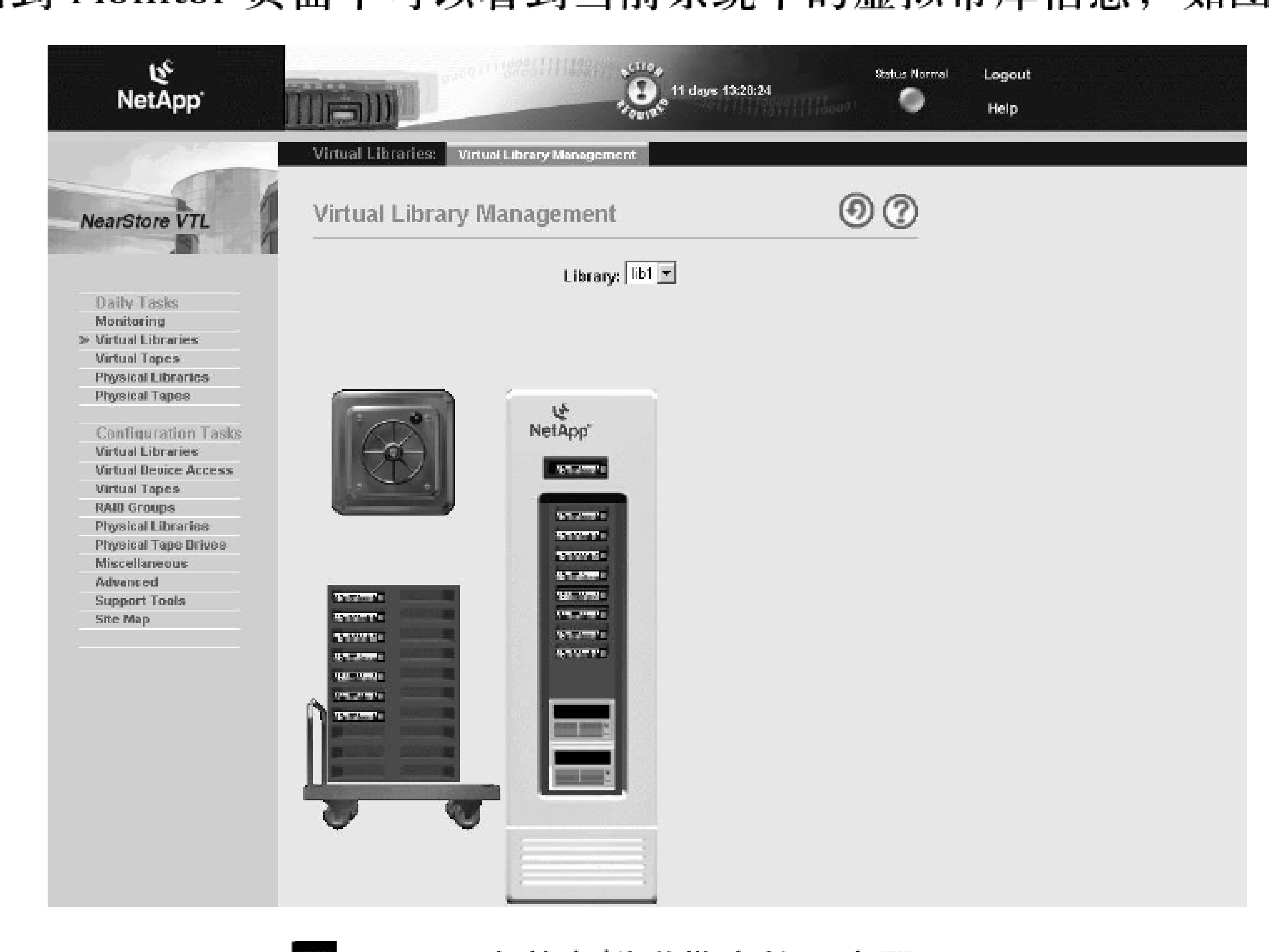


图14.20　当前虚拟磁带库的示意图页面

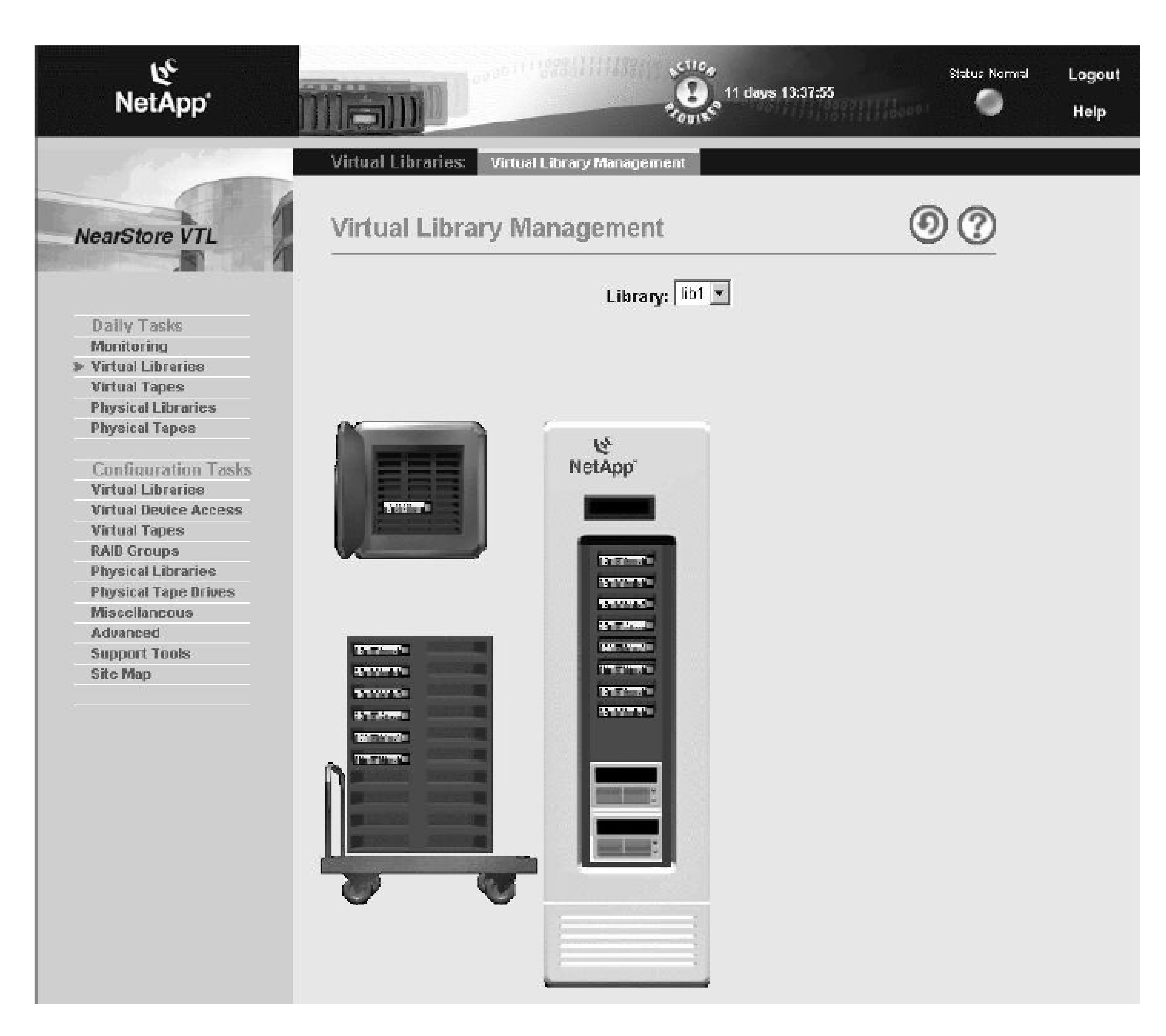


图14.21 删除虚拟磁带

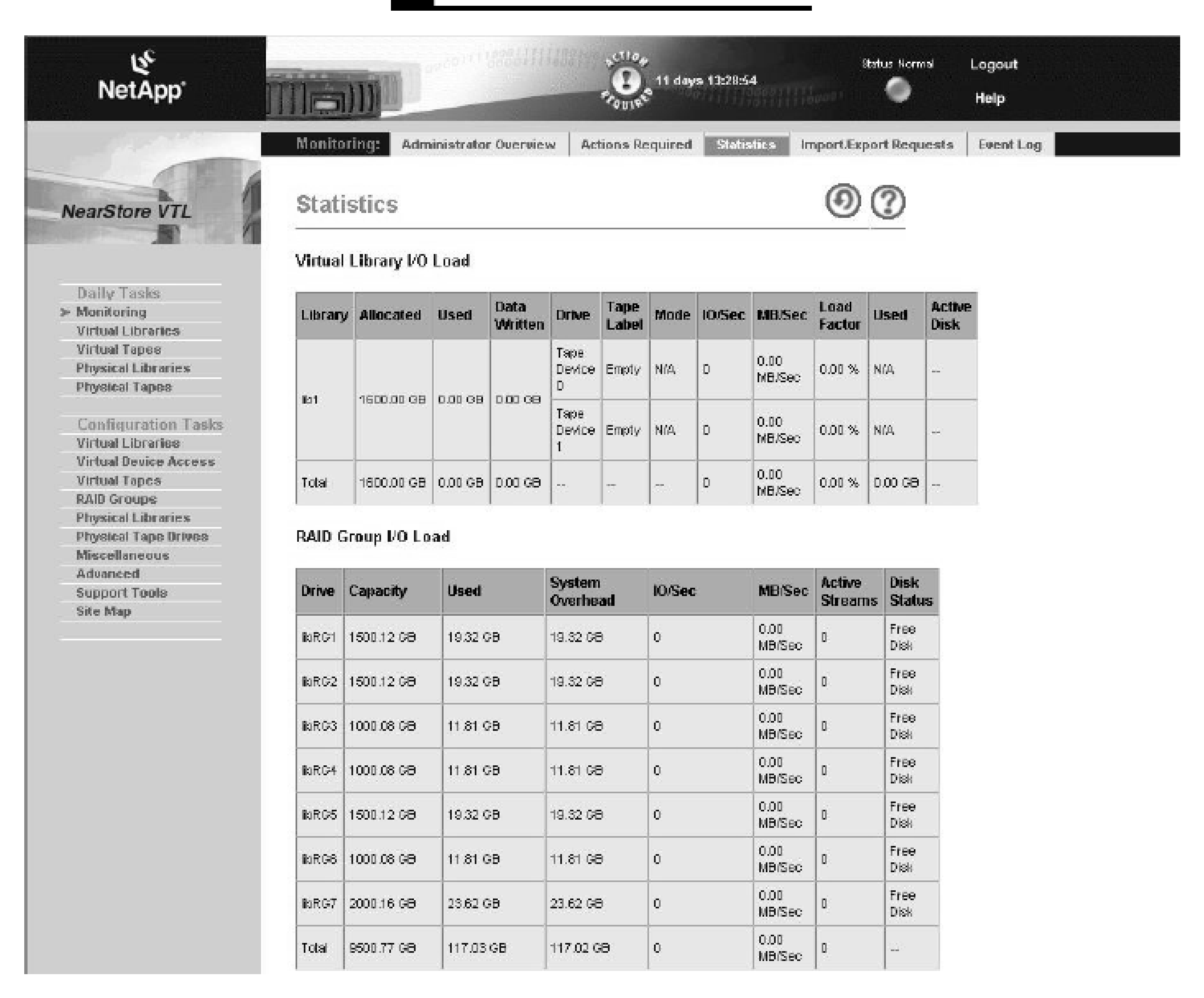


图14.22 当前系统的虚拟带库信息

5】 创建第二台虚拟带库。再次创建一台名为“lib2”的虚拟带库。这里选择驱动器类型为“HP LTO1”，16 槽位，4 驱动器，如图 14.23 所示。图 14.24 显示了“lib2”的拓扑图，可以和“lib1”的图对比一下。

6】 单击左侧栏 Virtual Tapes 标签，右侧页面中可以查看虚拟磁带的信息、修改虚拟磁带的属性、在多台虚拟带库中移动磁带、删除虚拟磁带。如图 14.25～图 14.28 所示。



图14.23　创建“lib2”虚拟带库

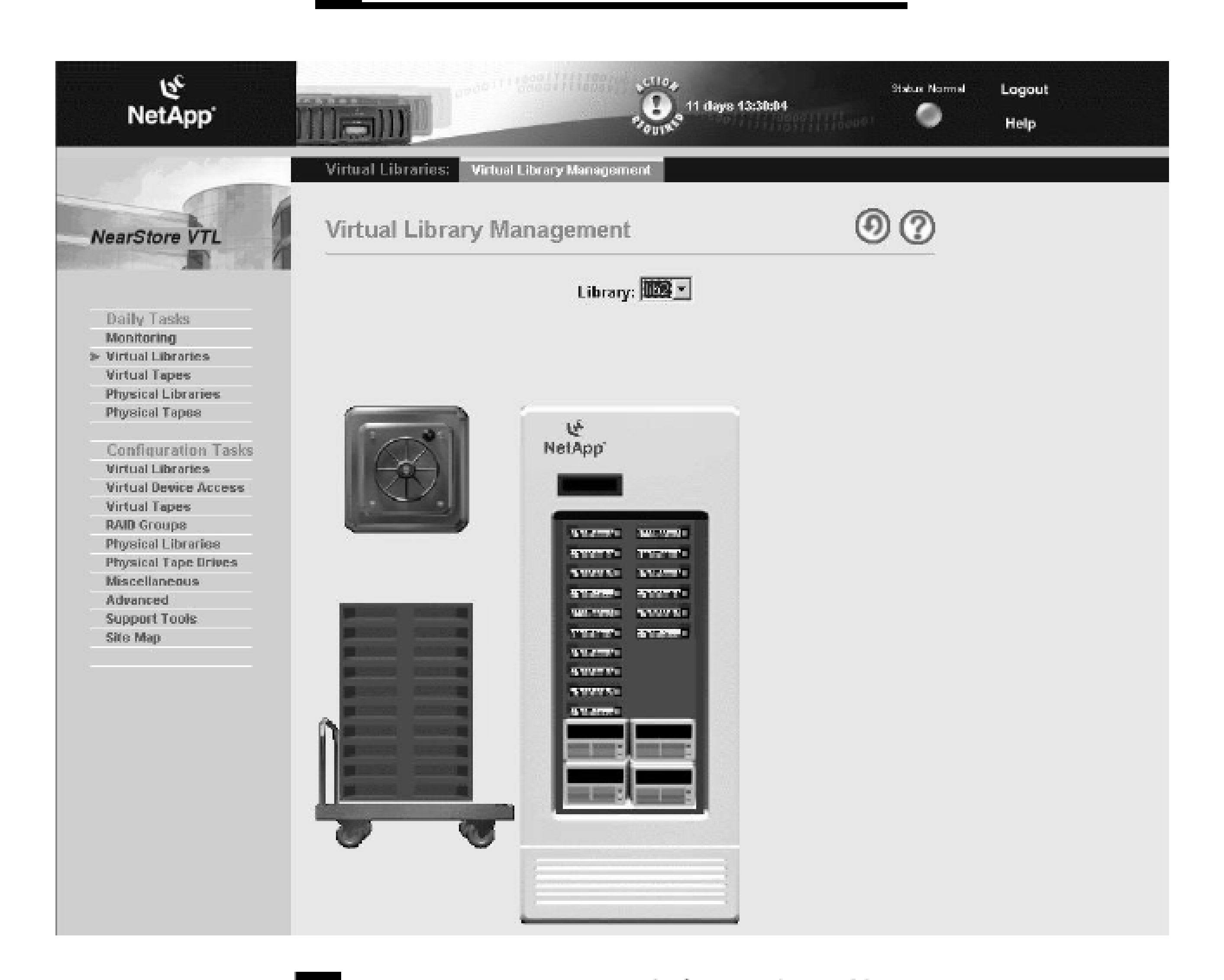


图14.24　“lib2”虚拟带库拓扑图

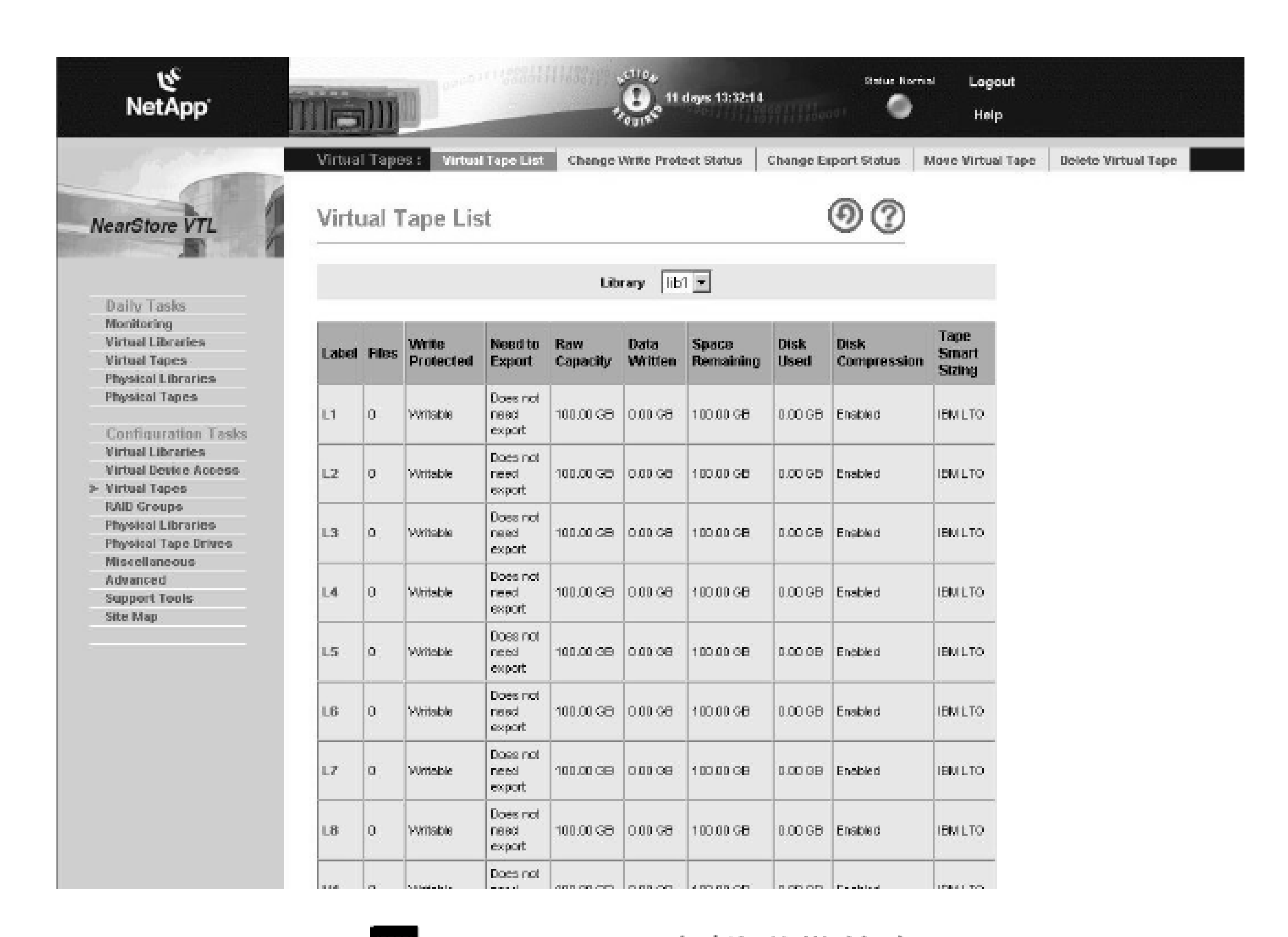


图14.25　显示虚拟磁带信息

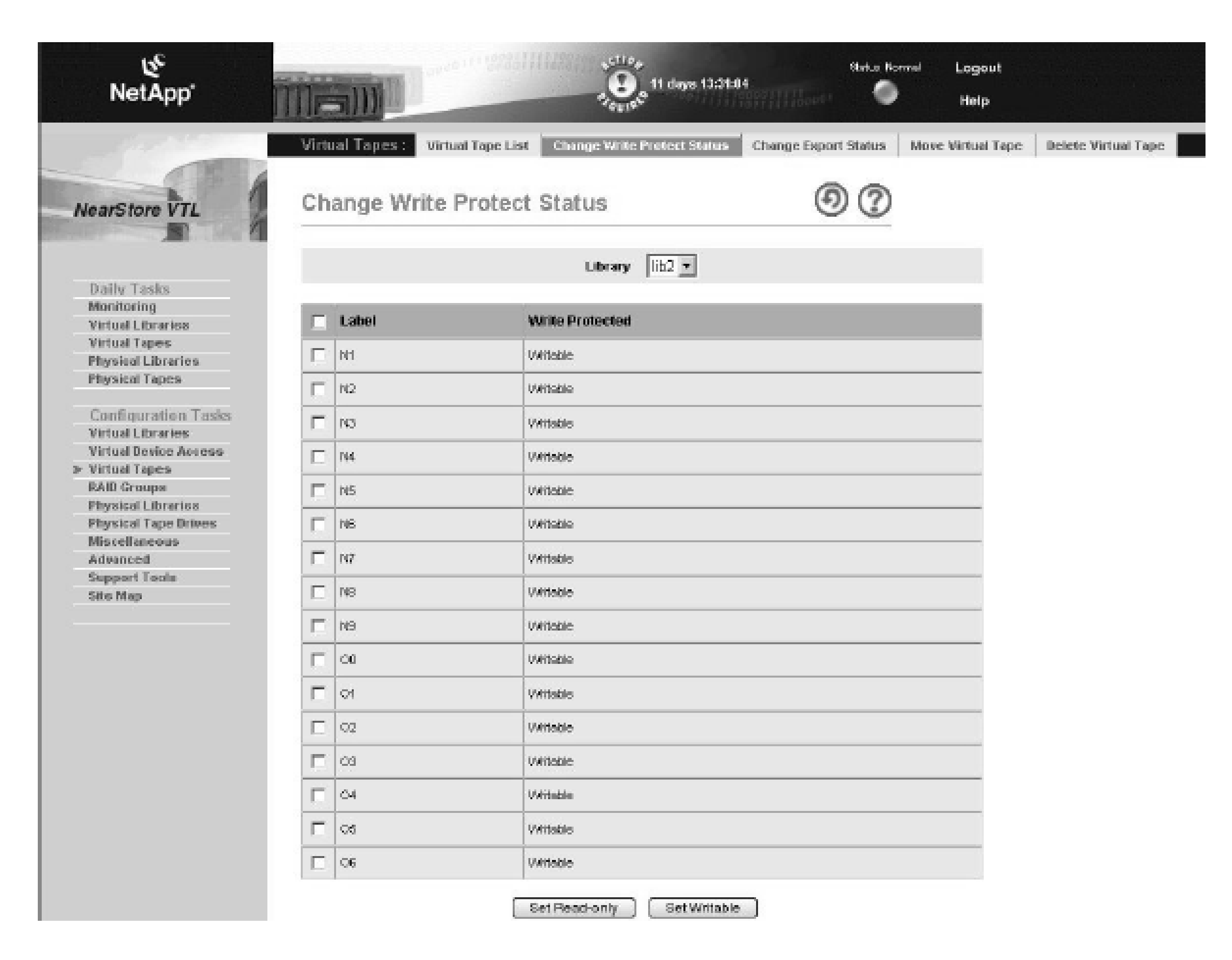


图14.26 修改虚拟磁带的读写属性

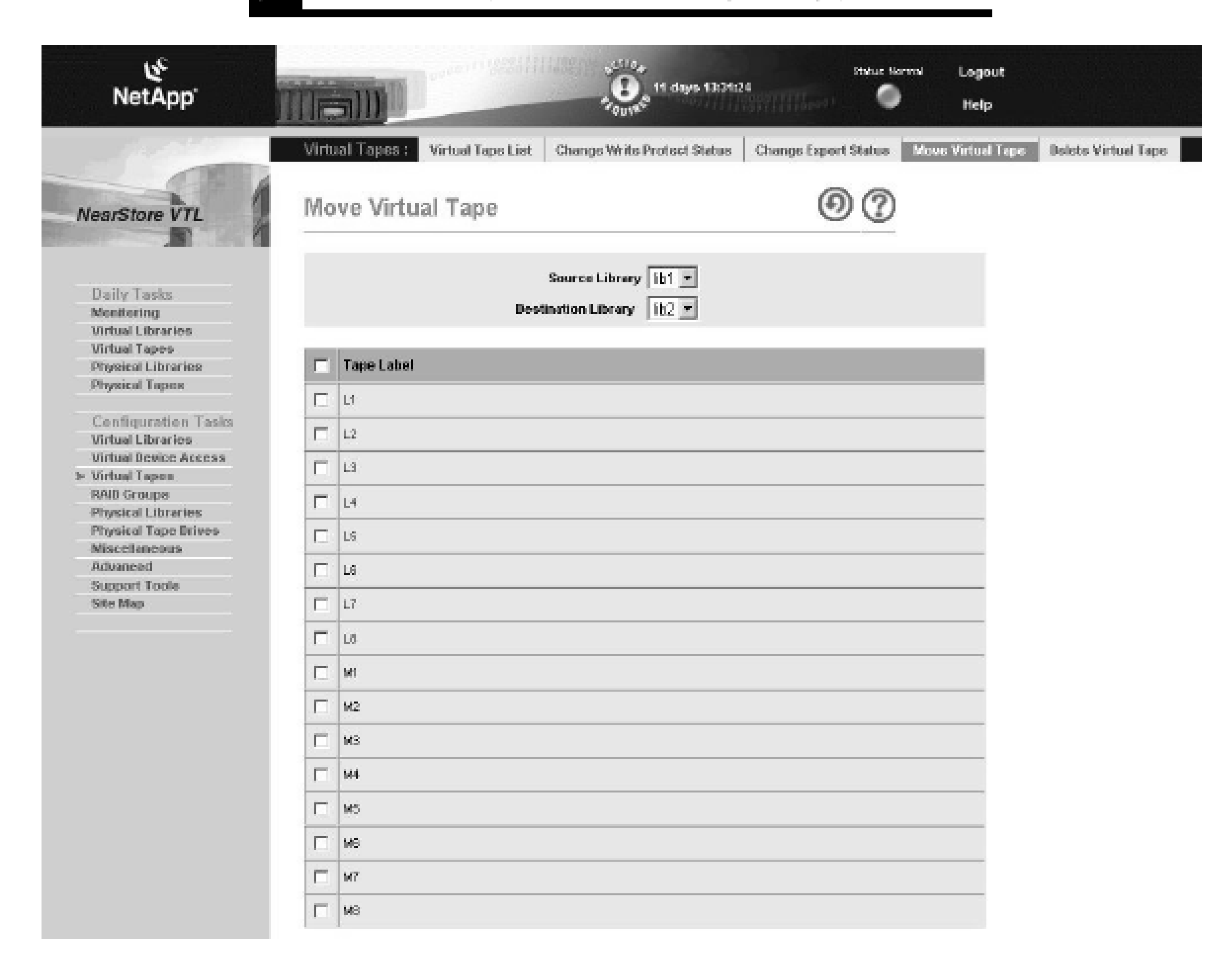


图14.27 在多台虚拟带库之间移动磁带

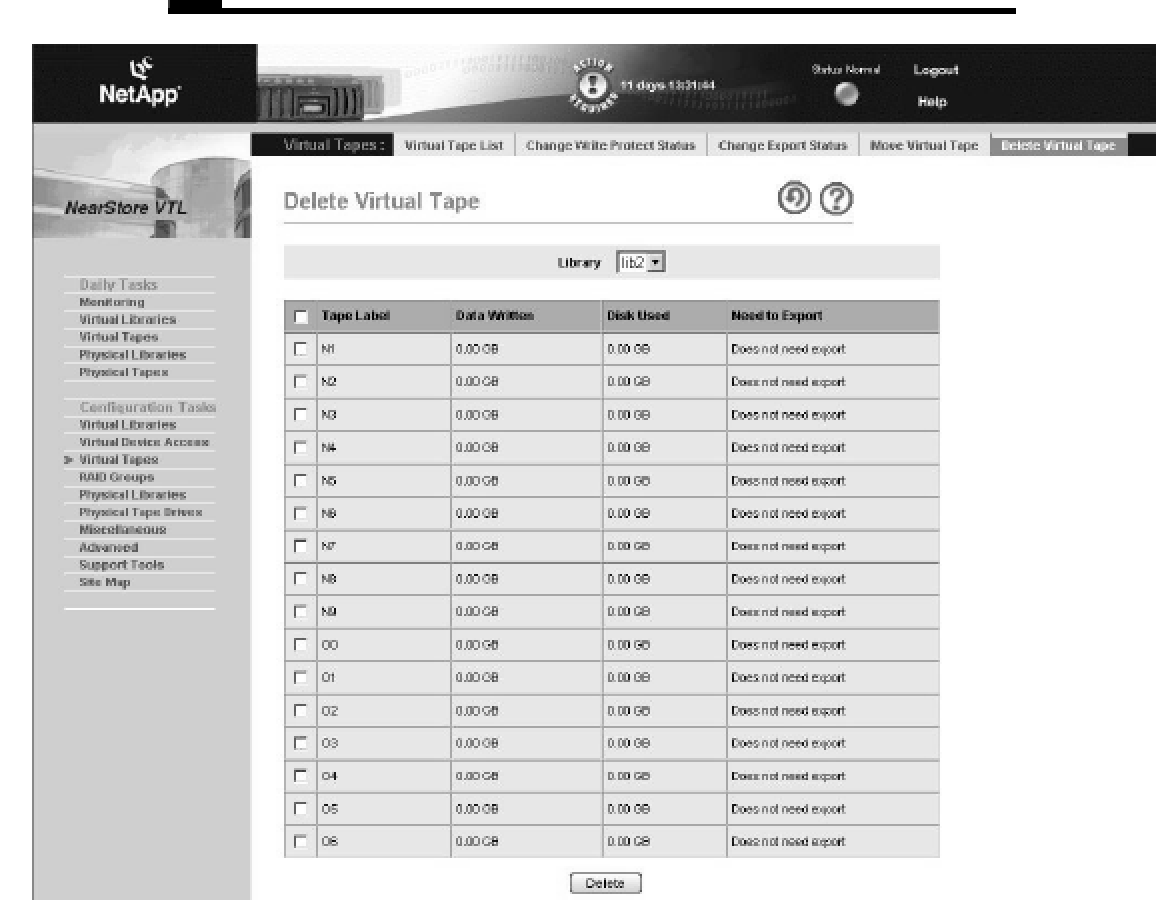


图14.28 删除虚拟磁带

7】 返回 Monitor 查看页面信息，如图 14.29 所示，出现另一台虚拟带库。

8】 手动将磁带放入驱动器。如图 14.30 所示，可以在拓扑图中拖动任何一盘磁带进

入驱动器。这个动作就好比在物理带库中，由机械手将磁带从磁带槽中抓出，并推入磁带驱动器。当然，备份软件可以发送指令让带库自动做这个动作，我们当然也可以手动做这个动作。磁带放入驱动器之后，Monitor 页面中会显示出当前带库的驱动器中所包含的磁带，如图 14.31 所示。

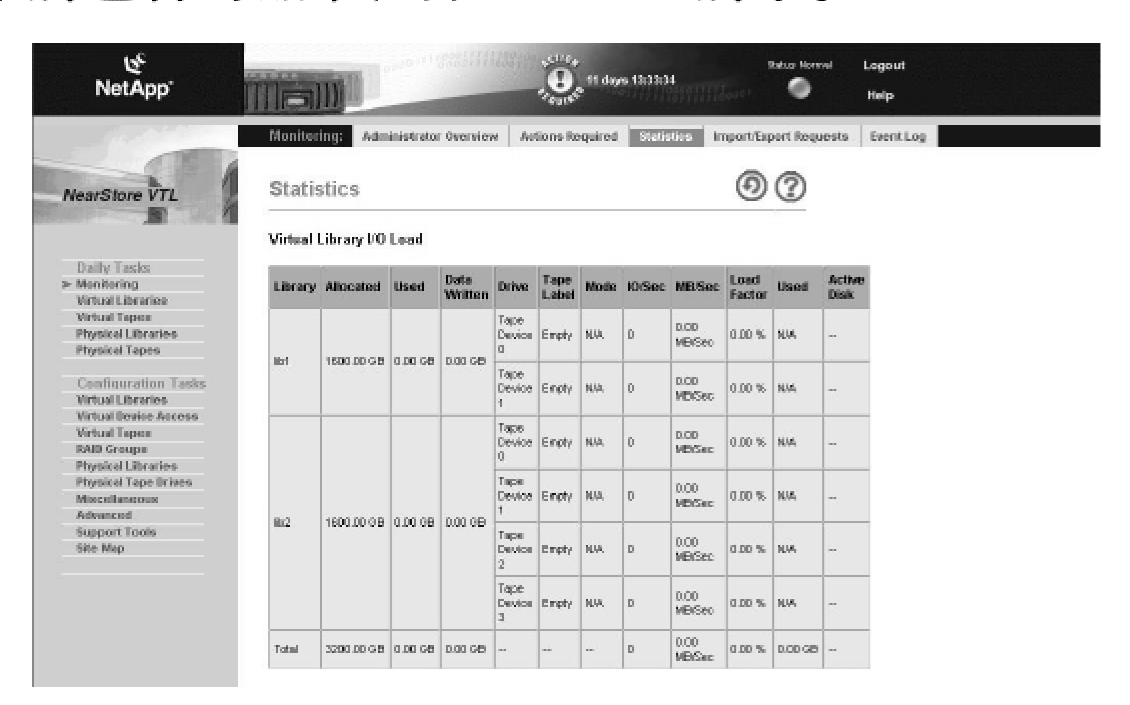


图14.29　系统中的两台虚拟带库

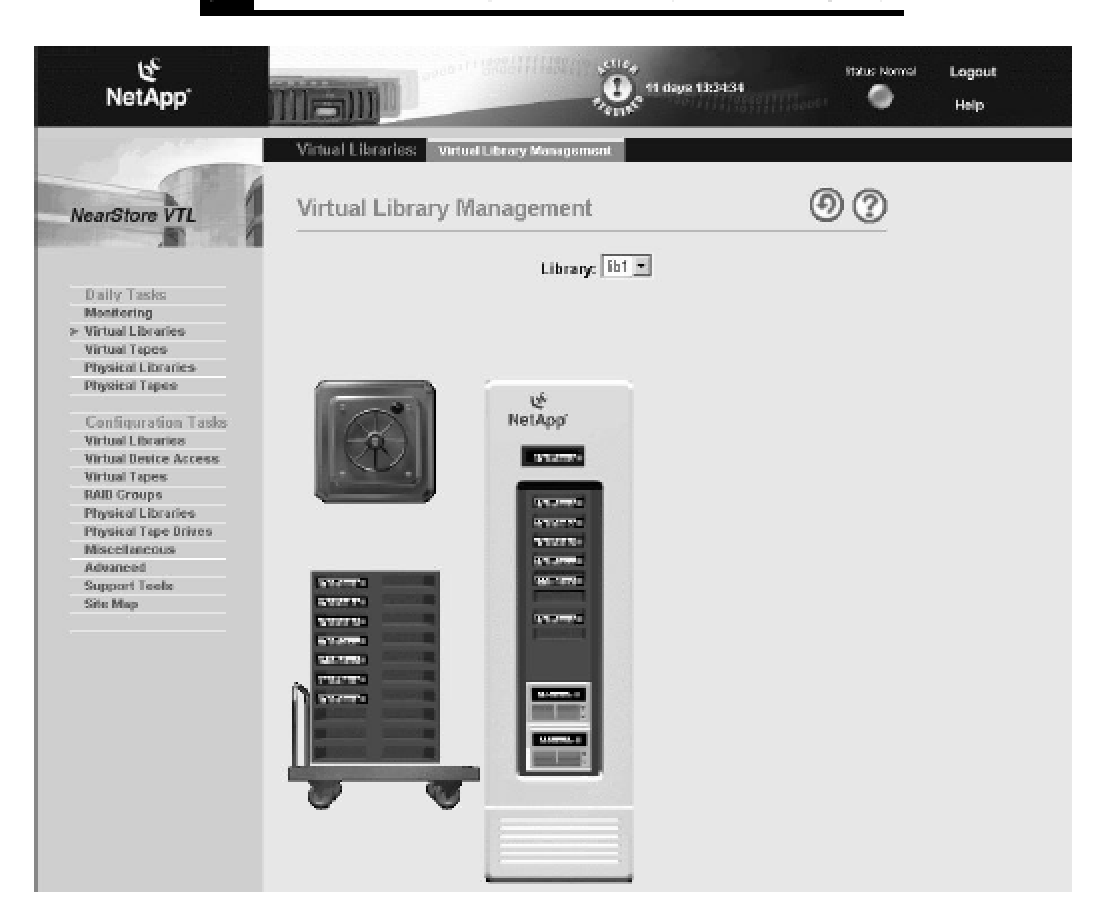


图14.30　手动将磁带放入驱动器



图14.31　Monitor 页面中显示的驱动器状态

3. 第三步：在客户端使用虚拟磁带库

客户端对 VTL 的使用与使用一台纯物理磁带库没有任何区别。我们用一台 NetApp 的 FAS3050 磁盘阵列来识别这台 VTL，看看效果。在 FAS3050 命令行中输入“sysconfig –a”来查看当前系统中的所有设备，可以看到在 FC 通道“0c”上已经识别到了 2 个 IBM 的驱动

器和 4 个 HP 的驱动器，这和 VTL700 上的配置完全一样，如图 14.32 所示。输入“storage show tape”命令可以查看更详细的磁带驱动器信息，如图 14.33 所示。输入“storage show mc”命令可以查看识别到的机械手信息(机械手是 VTL 虚拟出来的)，如图 14.34 所示。输入“sysconfig –t”命令可以查看系统识别到的所有磁带，如图 14.35 所示。

```
slot 0: FC Host Adapter 0c (QLogic 2432 rev. 2, L-port, <UP>)
        Firmware rev:      4.0.24
        Host Loop Id:      7
        FC Node Name:      5:00a:098000:01ae83
        SFP Vendor:        FINISAR CORP.
        SFP Part Number:   FTLF8524P2BNV
        SFP Serial Number: PB90F51
        SFP Capabilities:  1, 2 or 4 Gbit
        Link Data Rate:    4 Gbit
        0: Medium Changer: NETAPP  VTL                0001
        0L3: Medium Changer: NETAPP  VTL                0001
        0L1: Tape: IBM     ULTRIUM-TD1      022C
        0L2: Tape: IBM     ULTRIUM-TD1      022C
        0L4: Tape: HP      Ultrium 1-SCSI   022C
        0L5: Tape: HP      Ultrium 1-SCSI   022C
        0L6: Tape: HP      Ultrium 1-SCSI   022C
        0L7: Tape: HP      Ultrium 1-SCSI   022C
                I/O base 0x0, size 0x0
                memory mapped I/O base 0xc1300000, size 0x4000
```

图14.32　客户端所识别到的磁带设备

```
fas3040cl1-cn> storage show switch
No switches found.
fas3040cl1-cn> storage show tape

Tape Drive:           0c.0L1
Description:          IBM     ULTRIUM-TD1
Serial Number:        be6f99663f81a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L1
Alias Name(s):        st0
Device State:         available

Tape Drive:           0c.0L2
Description:          IBM     ULTRIUM-TD1
Serial Number:        be6f9f743f81a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L2
Alias Name(s):        st1
Device State:         available

Tape Drive:           0c.0L4
Description:          HP      Ultrium 1-SCSI
Serial Number:        1cf372f03f82a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L4
Alias Name(s):        st2
Device State:         available

Tape Drive:           0c.0L5
Description:          HP      Ultrium 1-SCSI
Serial Number:        1cf378223f82a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L5
Alias Name(s):        st3
Device State:         available

Tape Drive:           0c.0L6
Description:          HP      Ultrium 1-SCSI
Serial Number:        1cf37c823f82a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L6
Alias Name(s):        st4
Device State:         available

Tape Drive:           0c.0L7
Description:          HP      Ultrium 1-SCSI
Serial Number:        1cf3806a3f82a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L7
Alias Name(s):        st5
Device State:         available

fas3040cl1-cn>
```

图14.33　详细的驱动器信息

```
fas3040cl1-cn> storage show mc

Medium Changer:       0c.0
Description:          NETAPP   VTL
Serial Number:        be6f92723f81a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]
Alias Name(s):        mc0
Device State:         available

Medium Changer:       0c.0L3
Description:          NETAPP   VTL
Serial Number:        1cf366483f82a0980821b4
World Wide Name:      WWN[5:00a:098200:01aeb1]L3
Alias Name(s):        mc1
Device State:         available

fas3040cl1-cn> █
```

图14.34　查看机械手信息

```
fas3040cl1-cn*> sysconfig -t

    Tape drive (0c.0L1)  IBM     ULTRIUM-TD1
    rst0l  -  rewind device,        format is: LTO Format 100 GB
    nrst0l -  no rewind device,     format is: LTO Format 100 GB
    urst0l -  unload/reload device, format is: LTO Format 100 GB
    rst0m  -  rewind device,        format is: LTO Format 100 GB
    nrst0m -  no rewind device,     format is: LTO Format 100 GB
    urst0m -  unload/reload device, format is: LTO Format 100 GB
    rst0h  -  rewind device,        format is: LTO Format 100 GB
    nrst0h -  no rewind device,     format is: LTO Format 100 GB
    urst0h -  unload/reload device, format is: LTO Format 100 GB
    rst0a  -  rewind device,        format is: LTO Format 200 GB comp
    nrst0a -  no rewind device,     format is: LTO Format 200 GB comp
    urst0a -  unload/reload device, format is: LTO Format 200 GB comp

    Tape drive (0c.0L2)  IBM     ULTRIUM-TD1
    rst1l  -  rewind device,        format is: LTO Format 100 GB
    nrst1l -  no rewind device,     format is: LTO Format 100 GB
    urst1l -  unload/reload device, format is: LTO Format 100 GB
    rst1m  -  rewind device,        format is: LTO Format 100 GB
    nrst1m -  no rewind device,     format is: LTO Format 100 GB
    urst1m -  unload/reload device, format is: LTO Format 100 GB
    rst1h  -  rewind device,        format is: LTO Format 100 GB
    nrst1h -  no rewind device,     format is: LTO Format 100 GB
    urst1h -  unload/reload device, format is: LTO Format 100 GB
    rst1a  -  rewind device,        format is: LTO Format 200 GB comp
    nrst1a -  no rewind device,     format is: LTO Format 200 GB comp
    urst1a -  unload/reload device, format is: LTO Format 200 GB comp

    Tape drive (0c.0L4)  HP      Ultrium 1-SCSI
    rst2l  -  rewind device,        format is: LTO Format 100 GB
    nrst2l -  no rewind device,     format is: LTO Format 100 GB
    urst2l -  unload/reload device, format is: LTO Format 100 GB
    rst2m  -  rewind device,        format is: LTO Format 100 GB
    nrst2m -  no rewind device,     format is: LTO Format 100 GB
    urst2m -  unload/reload device, format is: LTO Format 100 GB
    rst2h  -  rewind device,        format is: LTO Format 100 GB
    nrst2h -  no rewind device,     format is: LTO Format 100 GB
    urst2h -  unload/reload device, format is: LTO Format 100 GB
    rst2a  -  rewind device,        format is: LTO Format 200 GB comp
    nrst2a -  no rewind device,     format is: LTO Format 200 GB comp
    urst2a -  unload/reload device, format is: LTO Format 200 GB comp

    Tape drive (0c.0L5)  HP      Ultrium 1-SCSI
    rst3l  -  rewind device,        format is: LTO Format 100 GB
    nrst3l -  no rewind device,     format is: LTO Format 100 GB
    urst3l -  unload/reload device, format is: LTO Format 100 GB
    rst3m  -  rewind device,        format is: LTO Format 100 GB
    nrst3m -  no rewind device,     format is: LTO Format 100 GB
    urst3m -  unload/reload device, format is: LTO Format 100 GB
    rst3h  -  rewind device,        format is: LTO Format 100 GB
    nrst3h -  no rewind device,     format is: LTO Format 100 GB
    urst3h -  unload/reload device, format is: LTO Format 100 GB
    rst3a  -  rewind device,        format is: LTO Format 200 GB comp
    nrst3a -  no rewind device,     format is: LTO Format 200 GB comp
    urst3a -  unload/reload device, format is: LTO Format 200 GB comp
     Tape drive (0c.0L6)  HP      Ultrium 1-SCSI
     rst4l  -  rewind device,        format is: LTO Format 100 GB
     nrst4l -  no rewind device,     format is: LTO Format 100 GB
     urst4l -  unload/reload device, format is: LTO Format 100 GB
     rst4m  -  rewind device,        format is: LTO Format 100 GB
     nrst4m -  no rewind device,     format is: LTO Format 100 GB
     urst4m -  unload/reload device, format is: LTO Format 100 GB
     rst4h  -  rewind device,        format is: LTO Format 100 GB
     nrst4h -  no rewind device,     format is: LTO Format 100 GB
     urst4h -  unload/reload device, format is: LTO Format 100 GB
     rst4a  -  rewind device,        format is: LTO Format 200 GB comp
     nrst4a -  no rewind device,     format is: LTO Format 200 GB comp
     urst4a -  unload/reload device, format is: LTO Format 200 GB comp

     Tape drive (0c.0L7)  HP      Ultrium 1-SCSI
     rst5l  -  rewind device,        format is: LTO Format 100 GB
     nrst5l -  no rewind device,     format is: LTO Format 100 GB
     urst5l -  unload/reload device, format is: LTO Format 100 GB
     rst5m  -  rewind device,        format is: LTO Format 100 GB
     nrst5m -  no rewind device,     format is: LTO Format 100 GB
     urst5m -  unload/reload device, format is: LTO Format 100 GB
     rst5h  -  rewind device,        format is: LTO Format 100 GB
     nrst5h -  no rewind device,     format is: LTO Format 100 GB
     urst5h -  unload/reload device, format is: LTO Format 100 GB
     rst5a  -  rewind device,        format is: LTO Format 200 GB comp
     nrst5a -  no rewind device,     format is: LTO Format 200 GB comp
     urst5a -  unload/reload device, format is: LTO Format 200 GB comp
fas3040cl1-cn*>
```

图14.35　系统所识别到的所有磁带

至此，这台 VTL 虚拟出了两台带库，当然还可以虚拟更多的带库，将它们分配到另外的 FC 端口。更加灵活的是，VTL 还可以自身连接物理磁带库，然后将这些物理资源透传到主机端，这样即使原来存在的物理带库也没有浪费，一起整合了进来。

各个厂家的 VTL 产品的设计都是大同小异，几乎都是用各自已经成形的盘阵产品，将其上运行的程序换一下，就变成了 VTL。图 14.36 是 EMC 公司的 VTL 产品的配置界面。可以看到各个厂家的设计都大同小异，本质都是一样的。

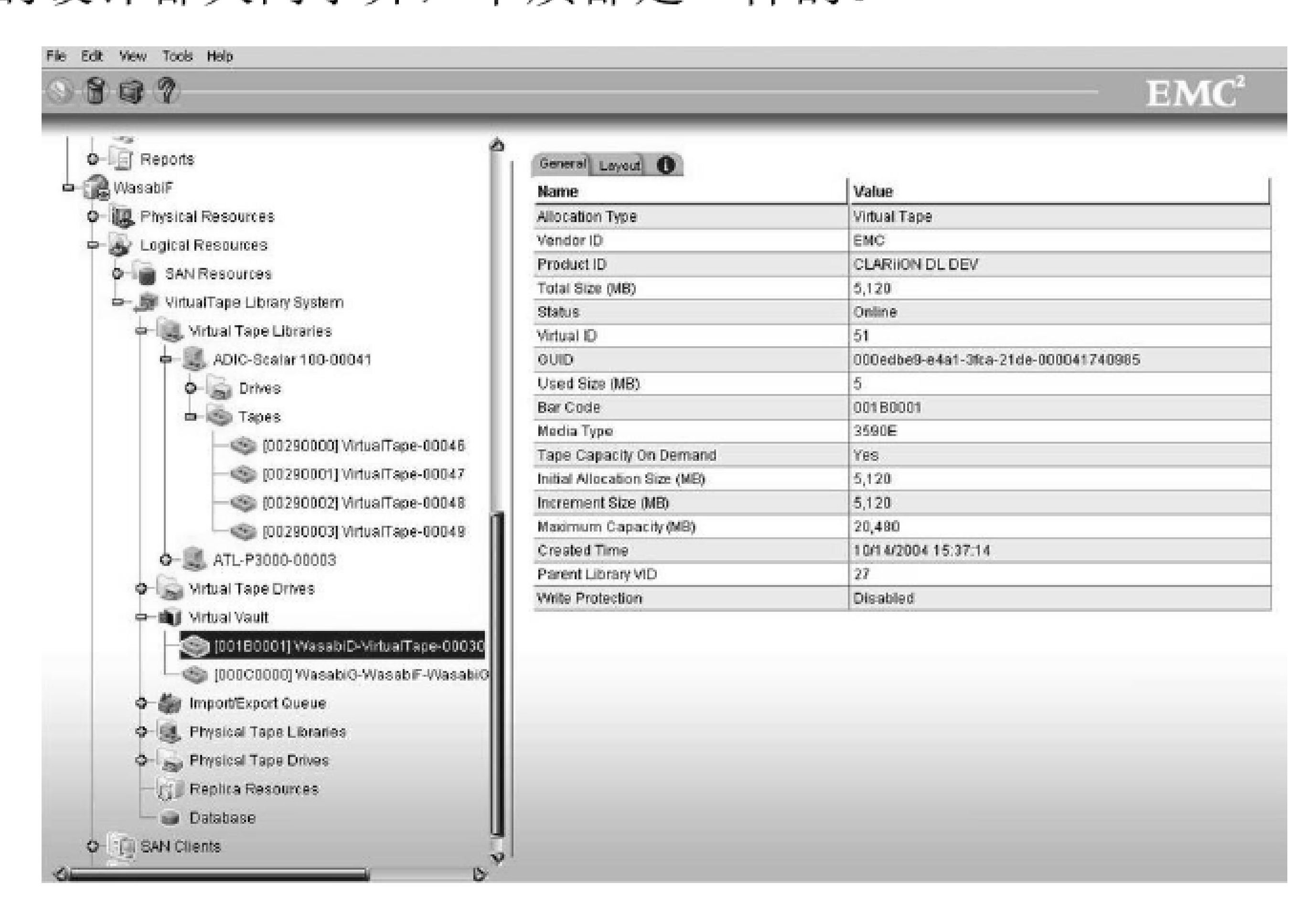


图 14.36　EMC 公司的 VTL 产品配置界面



存储群集

- 分布式
- 群集
- 高可用性群集
- 负载均衡群集
- 高性能群集

随着应用程序对服务器和存储系统的要求越来越高，对于传统设备来说，比如PC、PC服务器、小型机服务器等，单台设备有时已经不能满足需求了。此时虽然可以使用大型机，在单一设备上提供更高的性能，但是大型机的物质成本和维护成本是高不可攀的，而且大型机也不见得适合所有应用。怎么办呢？众人拾柴火焰高，人们想出了一种办法来应对日益扩张的应用程序需求，就是用多台设备联合起来对外提供服务，这就是群集。

主机可以形成群集，存储设备一样可以形成群集。目前中高端存储设备其自身就具备双控制器。不但如此，有一些NAS设备还可以在众多台独立设备之间形成群集，并且实现了单一名称空间，即用户访问目录路径像访问一台机器一样，而实际上，可能是由群集中不同的节点来提供服务。

15.1 群集概述

用多个节点来代替一个节点完成任务，毫无疑问是为了提高处理能力。其次，群集还可以做到高可用性，即一旦某个节点发生故障，不能再继续参与计算，那么群集中的其他节点可以立即接替故障节点的工作。

15.1.1 高可用性群集(HAC)

在 HA 群集中，节点分为活动节点和备份节点。活动节点就是正在执行任务的节点；备份节点是活动节点的备份。一旦活动节点发生故障，则备份节点立即接替活动节点来执行任务。高可用性群集的实现是基于资源切换的。所谓“资源”是指 HA 群集中某个节点发生故障之后，备份节点所要接管的任何东西的一个抽象的词汇。比如，在某个节点发生故障之后，其对应的备份节点，需要接管故障节点上的 IP 地址、主机名、磁盘卷、应用程序的上下文等，这样才能将对客户端造成的影响缩减到最小。这些被接管的实体，便被称为“资源”。资源的监控和接管，依靠于 HA 软件。目前存在多种 HA 软件。每种操作系统几乎都自带 HA 软件，它的作用就是监控对方节点的状态，一旦侦测到对方的任何故障，那么便会强行将所有资源占为已有并向客户端继续提供服务。

15.1.2 负载均衡群集(LBC)

在负载均衡群集中，群集中的所有节点都参与工作，每个节点的地位相同，接受的工作量按照某种策略，由一个单独的节点作为调度来向其他所有参与运算的节点分配，或者由所有参与运算的节点之间通过网络通信来协商分配。分配策略，比如轮流分配、随机分配、最小压力分配等。

15.1.3 高性能群集(HPC)

高性能群集，又称科学计算群集。这种群集其实与 LBC 群集的本质是相同的。只不过其专用于科学计算，即超大运算量的系统，比如地质勘探、气象预测、分子筛选、仿生模拟、蛋白质构型、分子药物分析、人工智能等。这些运算要么逻辑复杂、要么需要大量穷举，耗费极大 CPU 和内存资源。有些需要几天、几个月甚至半年才能执行完毕。此时，增加整个系统的 CPU 总核心数，可以成倍的缩短执行时间。



记得我在大学做毕业设计的时候，有个同学的课题就是计算分子式，这课题也简单，就是第一天将任务执行上，一个月之后结果出来了，写论文、答辩。那时候用的计算机都是 Intel 奔腾 4 的 CPU，倘若用现在酷睿多核 CPU，我想只需要十几天便可以出结果。

HPC 群集中，为了增加整个系统的 CPU 核心数，一般引入十几台或者几十台、几百台计算机，其中每台计算机又可以有多个物理 CPU，每个 CPU 又可以有多个核心。这样整个

系统的 CPU 核心数会相当可观。那么如何利用这么多的 CPU 呢？如何将任务平均分配到每个 CPU 核心上呢？

Windows 2000 以后的 Windows 系统，操作系统默认便自动支持同一台计算机内的多个 CPU 或者多个 CPU 核心，操作系统自动将多个线程平摊到多个 CPU 核心上运行。但是对于不处于同一台计算机内的 CPU 来说，任务将要怎么分配到其他节点上呢？当然是通过网络了。为了方便编程出现了很多 API，为程序员屏蔽掉多 CPU 所带来的编程复杂度，程序员只要按照这些 API 规范来编写代码，底层便会自动将运算任务分派到网络上的其他运算节点上。节点接收到任务数据之后，再由节点操作系统自行将这块任务数据分派到节点的多个 CPU 核心上。MPI 便是一个目前广泛应用的 HPC 系统 API。

15.2 群集的适用范围

群集可以实现在系统路径的任何点上。

- 硬件上：CPU、内存、显卡、显示终端、以太网卡、计算机本身、以太网及 IP 网络设备、FC 卡、FC 网络交换设备、磁盘阵列控制器本身、磁盘阵列控制器内部的各个组件、磁盘本身、磁盘内部的多片盘片和多个磁头。
- 软件上：应用程序、文件系统、卷管理系统。

什么时候需要实现群集

当某个系统的处理能力不能满足性能要求的时候，可考虑使用负载均衡群集或者高性能群集。当追求系统的高可用性时，即希望某处故障不会影响整个系统的可用性的时候，使用高可用性群集。当需要运算的数据量很大，运算周期很长的时候，可考虑实施高性能群集。

目前，各大知名网站一般都采用负载均衡群集来均衡 TCP 连接请求。由于这些网站每天的访问量很大，同时产生的 TCP 连接请求也很多，所以如果只用一台计算机来接受这些请求，根本满足不了性能，甚至会造成这台机器资源耗尽而死机。基于 Linux 系统的 LVS 负载均衡软件，是由国人主持研发的一种 TCP 负载均衡软件，被广泛用于 TCP 连接压力很大的系统下。LVS 可以基于很多策略来将前端的请求分摊到后端的多台计算机上。其本质就是一个基于策略的 TCP 包转发引擎。

对于比较重视 IT 建设的企业、重要的应用系统，都可实施 HA 群集来追求高可用性，从而避免故障造成的生产停顿。

各大科研院所、气象、石油勘探等机构，由于其需要很大的运算量和运算周期，一般都有 HPC 群集。

15.3 系统路径上的群集各论

15.3.1 硬件层面的群集

图 15.1 中箭头指向的部件都可以被群集化。

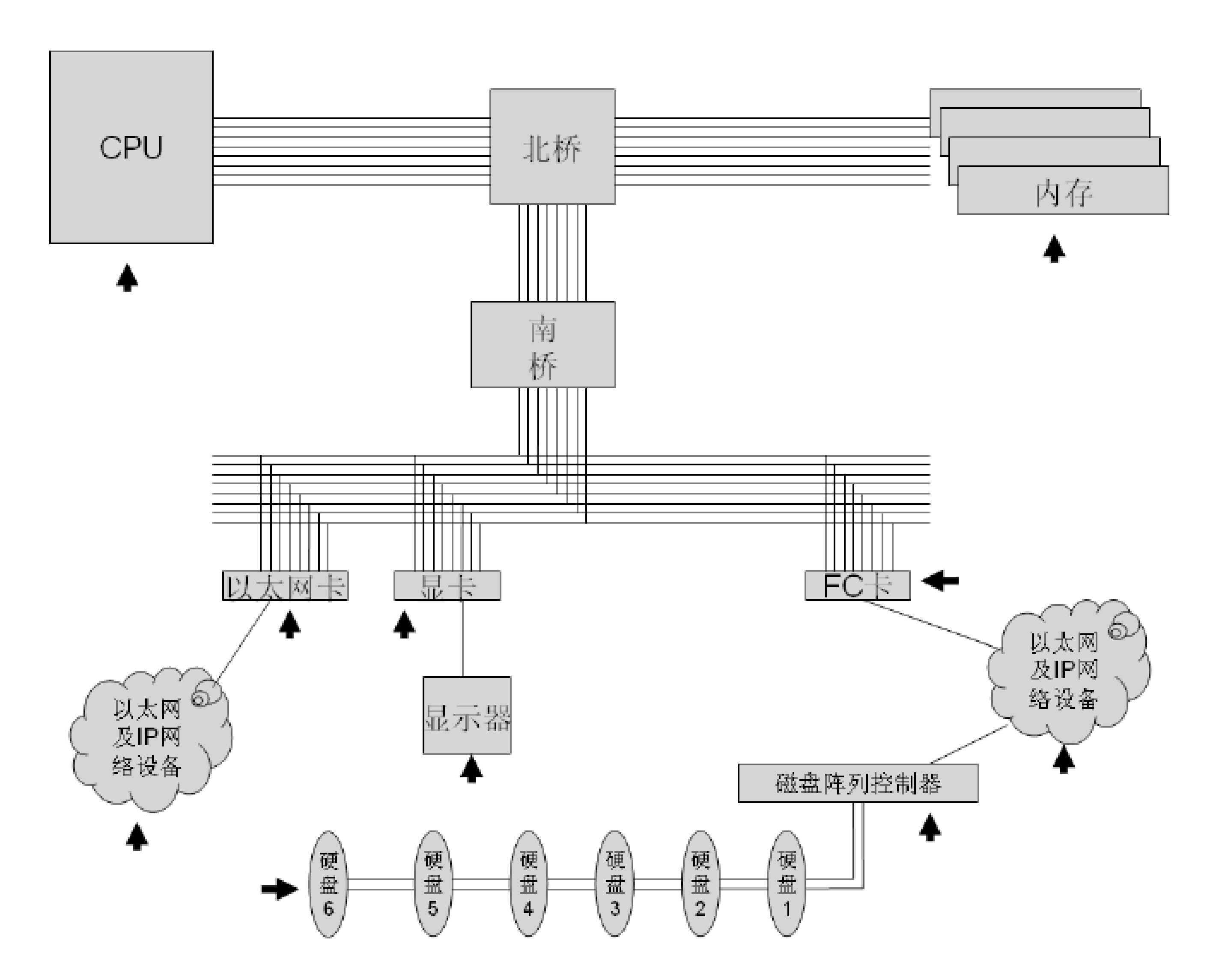


图15.1　系统路径上可实现群集的各处

- CPU 的群集。CPU 群集体现在多 CPU 的计算机系统，比如对称多处理器系统，多个 CPU 之间共享物理内存的共同协作。目前的服务器以及小型机系统大多为这种结构。
- 内存的群集。多条物理内存组成更大容量的空间，并且通过比如双通道(相当于磁盘系统中的条带化 RAID 0)等技术，提高性能。
- 以太网卡的群集。目前有多种方式来实现以太网卡的群集。将主机上的多块以太网卡绑定，向上层提供一块虚拟网卡，底层则可以通过 ARP 轮询负载均衡方式，或者 802.3ad 方式等向外提供负载均衡，或者 HA 方式的多路径访问。
- 以太网及 IP 网络设备的群集。在以太交换机和 IP 路由器上，多台设备之间协作转发网络数据包(帧)，诸如 Cisco、华为等厂商，都已经实现了负载均衡以及 HA 方式的群集。
- 显卡的群集。显卡群集是最近出现的技术。NVIDIA 以及 AMD 公司都有对应的解决方案。将插在总线上的多块显卡之间通过特殊连线连接起来，实现对大型 3D 数据渲染的负载均衡，性能得到很大提升。
- 显示器群集。比如电视墙等。但是这个严格来说并不算作群集。
- FC 卡的群集。通过与主机上的多路径软件配合，多块 FC 卡之间可以实现流量的负载均衡和 HA。或者通过 FC 网络中的 ISL 链路负载均衡、HA 方式实现流量分摊。
- FC 网络设备的群集。目前来说，FC 网络设备并没有像以太网以及 IP 网络设备那样实现负载均衡以及 HA。但是很多网络存储系统中，一般都部署多台 FC 交换机以避免单点故障，但是这个环境中的 FC 交换机本身并没有群集智能，所有群集逻辑都运行在 FC 节点上。
- 磁盘阵列控制器群集。目前几乎中高端的磁盘阵列的控制器都为双控架构，两个控制器之间可以为 HA 关系，或者为负载均衡关系。
- 磁盘的群集。典型的磁盘群集就是 RAID 系统，7 种 RAID(磁盘群集)方式，这里就不多描述了。其次磁盘内部的多块盘片，多个磁头之间也组成了群集，但这并不

能算作群集，因为同一时刻只能有一个磁头在读写。

15.3.2 软件层面的群集

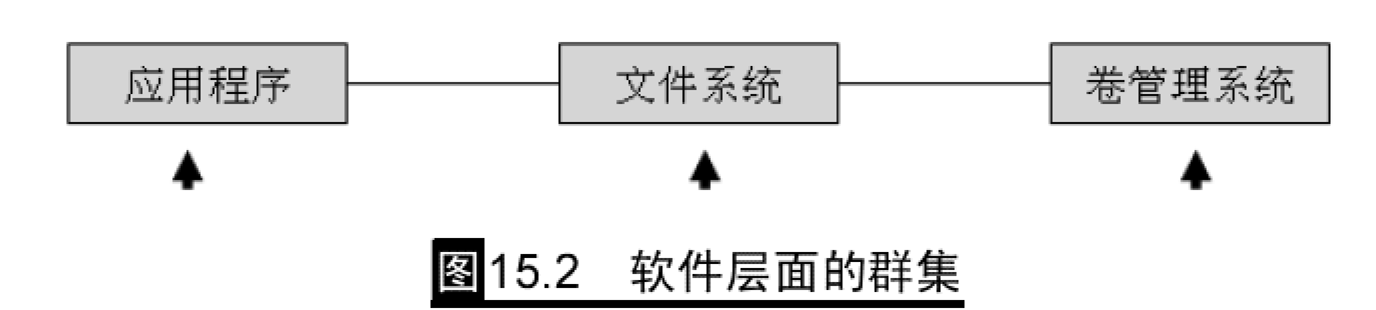


图15.2 软件层面的群集

1. 应用程序的群集

一个应用程序可以同时启动多个实例(进程)，共同完成工作。应用程序的不同实例可以运行在同一台机器上，也可以运行在不同的机器上，之间通过网络交互协商信息。

2. 文件系统的群集

文件系统的群集是一门比较独立的课题。可以实现群集功能的文件系统称为群集文件系统。比如 NFS、CIFS 等网络文件系统，就是最简单的群集文件系统。

群集文件系统的出现主要是为了解决三个问题：容量、性能、共享。

容量问题。群集文件系统有一类又被称为分布式文件系统。即某个全局目录下的存储空间，实际上是分布在群集中的各个节点上的。分布式文件系统将每个节点上的可用空间进行虚拟的整合，形成一个虚拟目录，并根据多种策略来判断数据的流向，从而将写入这个目录的数据对应成实际存储空间的写入。这样便可以做到群集中的整合存储，榨干最后一点群集的资源优势。

性能问题。用多个节点共同协作来获取高性能，这在文件系统层次依然成立。群集文件系统使得每个节点不必连接昂贵的磁盘阵列，就可以获得较高的文件 IO 性能。在分布式文件系统的虚拟整合目录的做法之上，又采取了类似磁盘条带 RAID 0 的处理方式，依据各种负载均衡策略，将每次 IO 写入的数据，分摊到所有节点上，节点获得的越多性能提升就越大。但这只是理论情况，实际使用起来群集文件系统并不是一个容易实施的系统，实施之后想要获得高性能，必须经过长时间的优化调试过程。

共享访问。群集文件系统所解决的最后一个问题，也是最为重要的一个问题，就是多节点共同访问相同目录和相同文件的问题。群集文件系统对多个节点，同时读写相同的文件做了很周全的考虑，能保证所有节点都能读到一致性的数据，并且利用分布式锁机制保证在允许的性能下，节点之间不会发生写冲突。

常见的群集文件系统有 PVFS、PVFS2、Lustre、GFS、GPFS、DFS、SANFS、SANergy 等。在这里就不做过多介绍了。

3. 卷管理系统的群集

本机的卷可以与本机卷或者远程计算机上的卷进行镜像等协同操作，形成群集。

15.4 实例：Microsoft MSCS 软件实现应用群集

Windows Server 2003 群集要求每台服务器上至少有两块以太网络适配器，一块作为公

用网络适配器(连接外部网络)，一块作为专用网络适配器(用于心跳检测)。群集中的所有节点必须在同一个域中，一般双机环境中直接使用其中一台为主域控制器，另一台为备份域控制器。

15.4.1 在 Microsoft Windows Server 2003 上安装 MSCS

1】 使用“控制面板”的“添加/删除程序”工具，添加 Windows 组件，安装群集服务。

2】 在管理工具菜单中打开群集管理器，当弹出群集连接向导时，选择“创建新群集”，并单击“下一步”按钮继续，如图 15.3 所示。

3】 输入群集的唯一 NetBIOS 名称(最多 15 个字符)，然后单击“下一步”按钮，如图 15.4 所示。

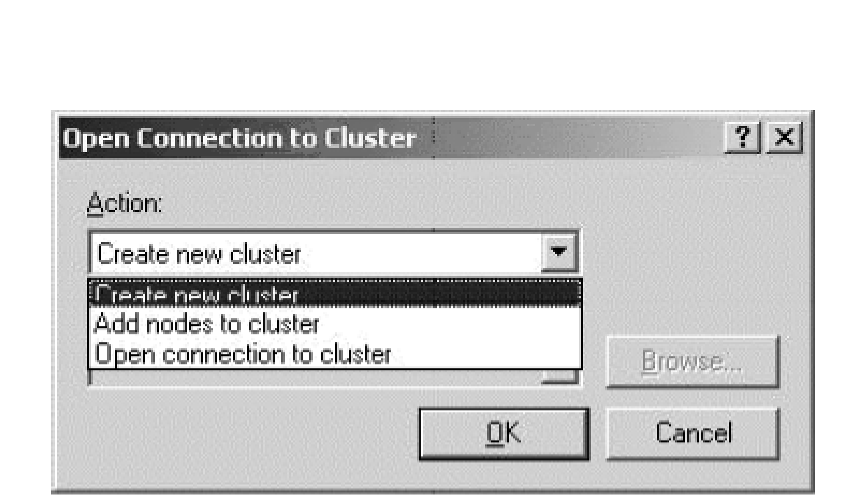


图15.3 创建新群集

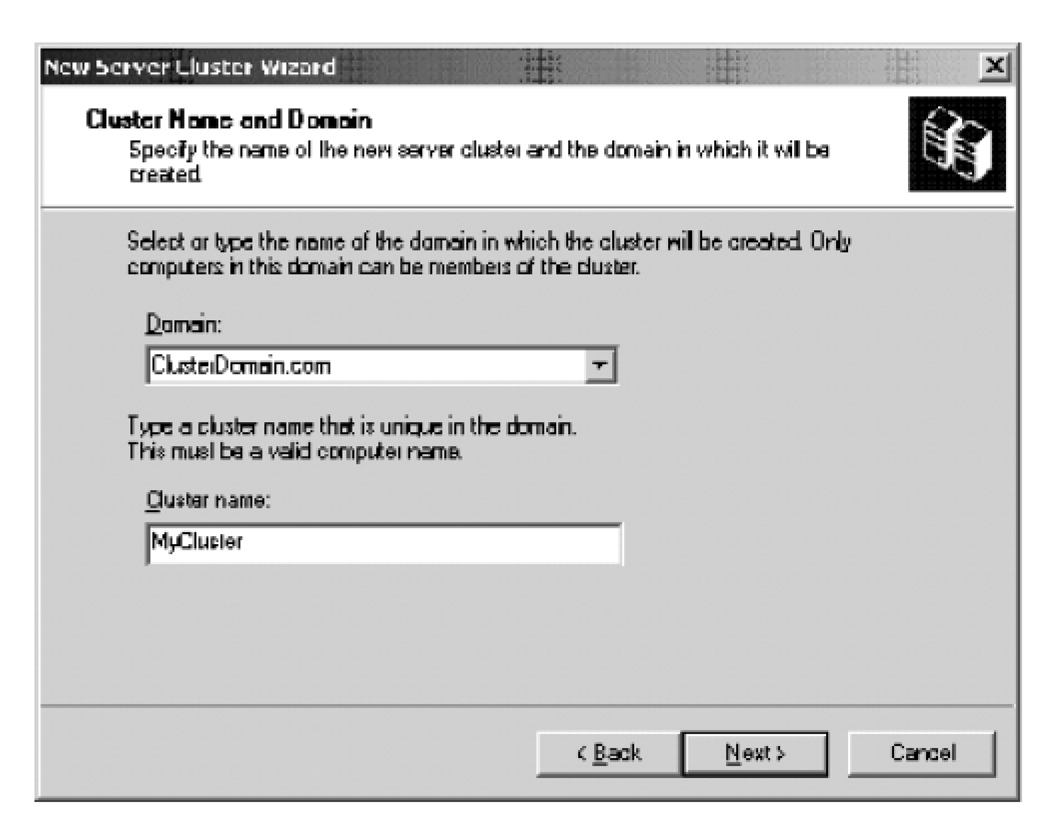


图15.4 输入群集名称

4】 如果在本地登录一个不属于“具有本地管理特权的域账户”的账户，向导会提示用户指定一个账户，如图 15.5 所示。

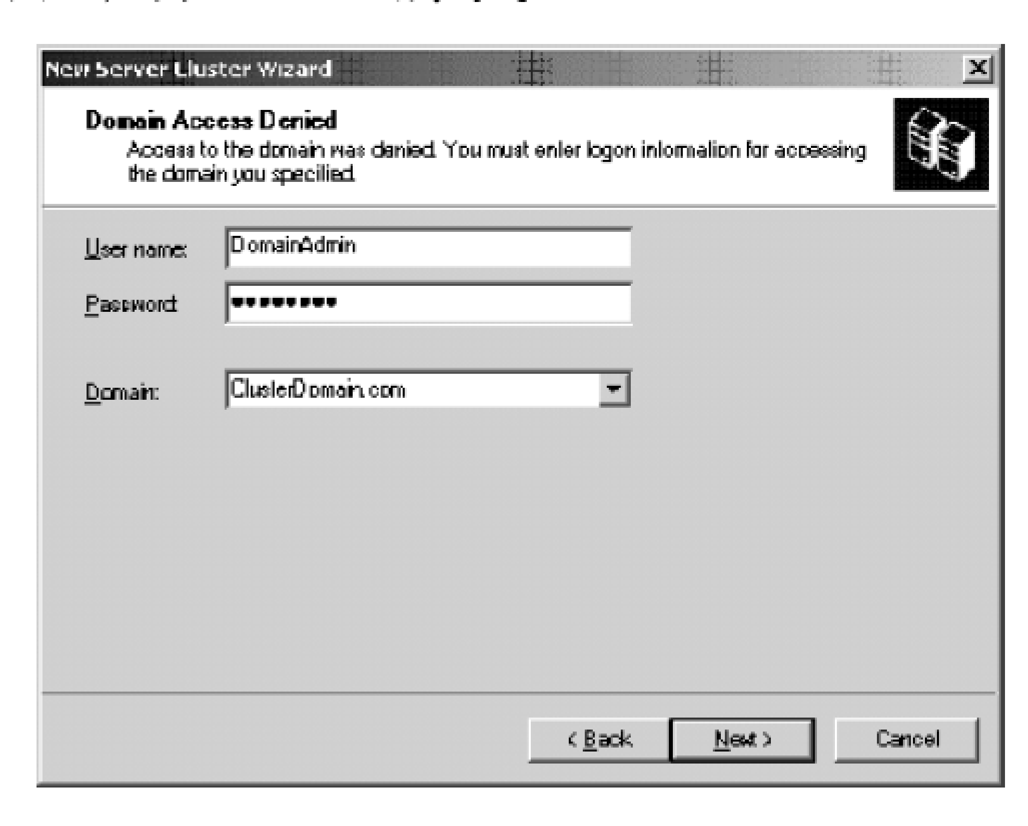


图15.5 输入群集账户信息

5】 确认将要作为第一个节点创建群集的服务器的名称，输入节点 1 的名称，如图 15.6 所示。

6】 安装程序将分析节点，查找可能导致安装出现问题的硬件或软件问题。检查所有

警告或错误信息。单击“详细信息”按钮，可以了解有关每个警告或提示的详细信息，如图 15.7 所示。

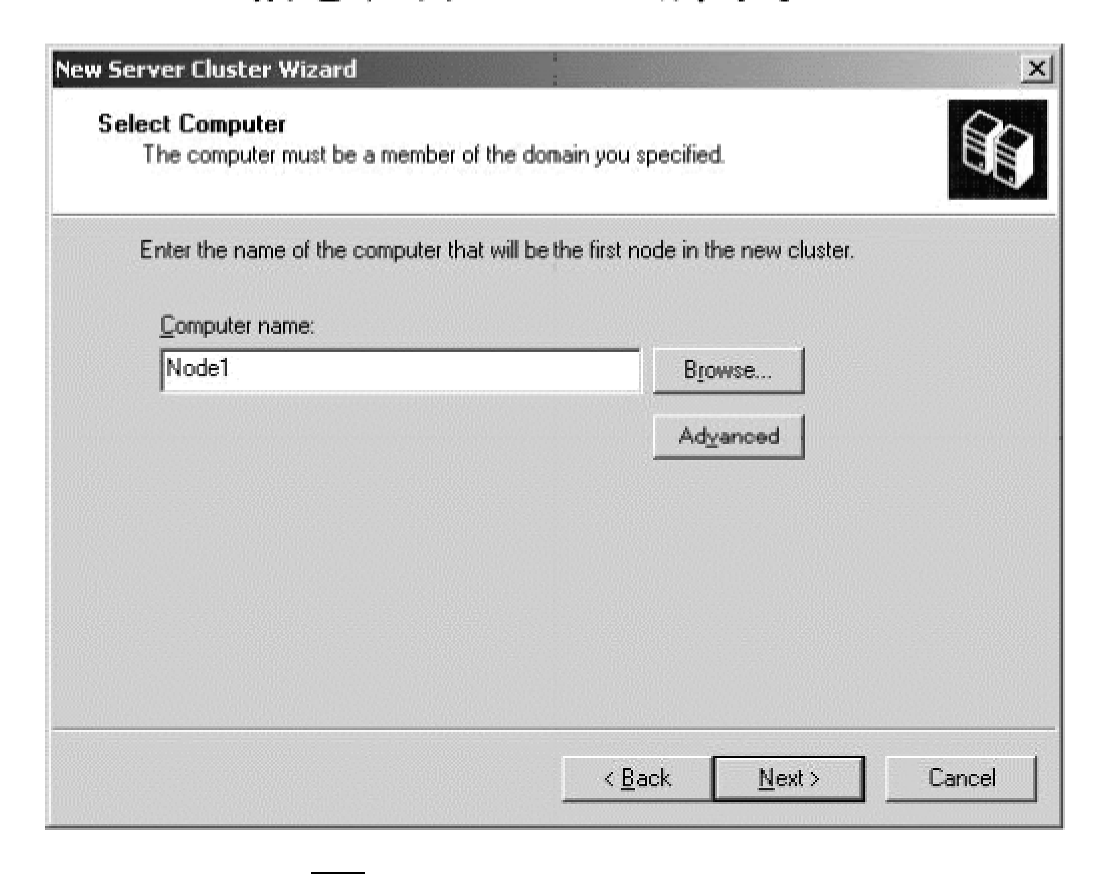


图15.6 输入节点名称

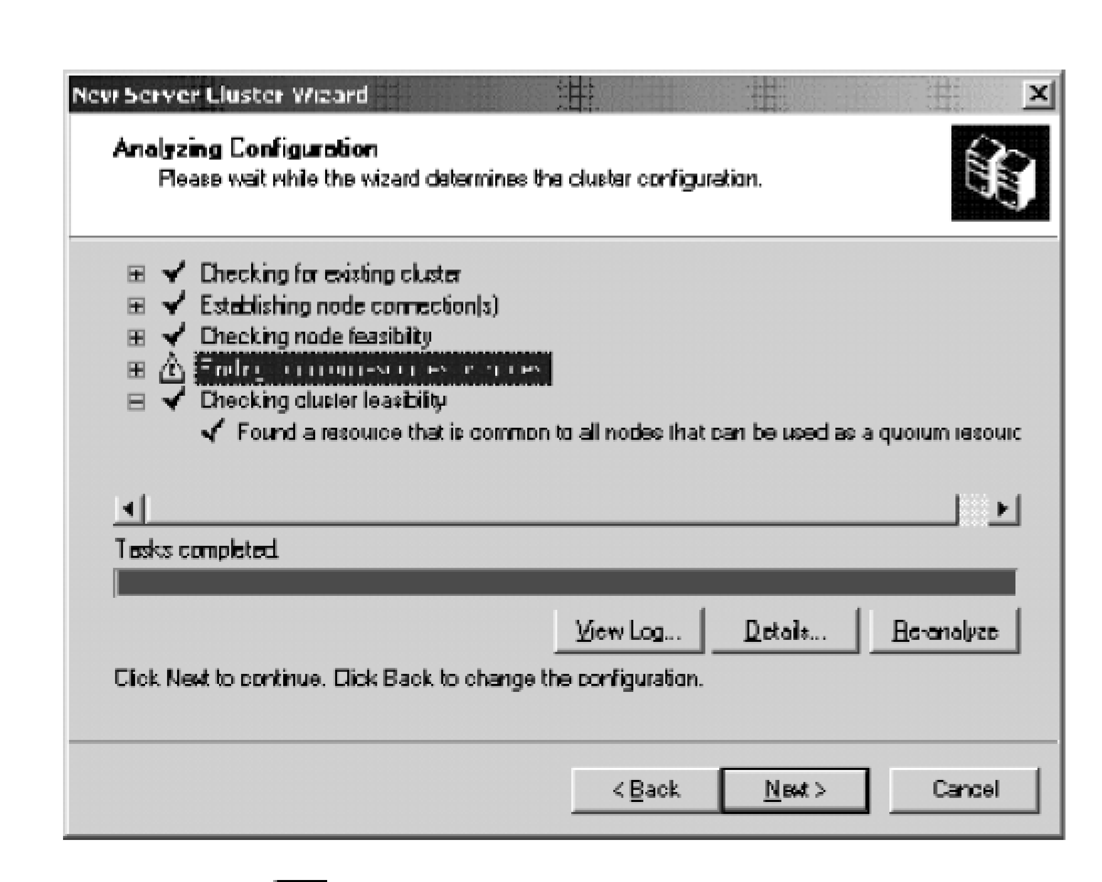


图15.7 检查群集配置环境

7】 输入唯一的群集 IP 地址。群集 IP 地址只能用于管理，而不能用于客户端连接，如图 15.8 所示。

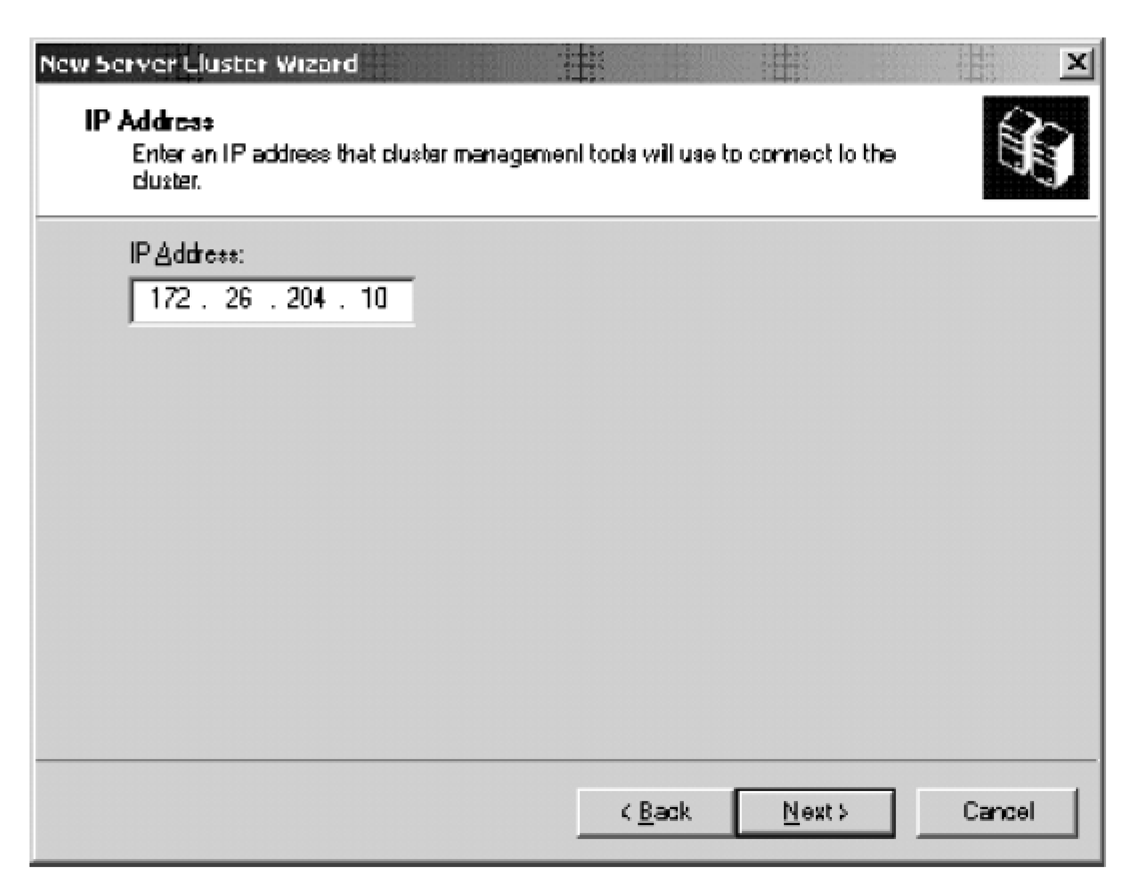


图15.8 输入群集 IP 地址

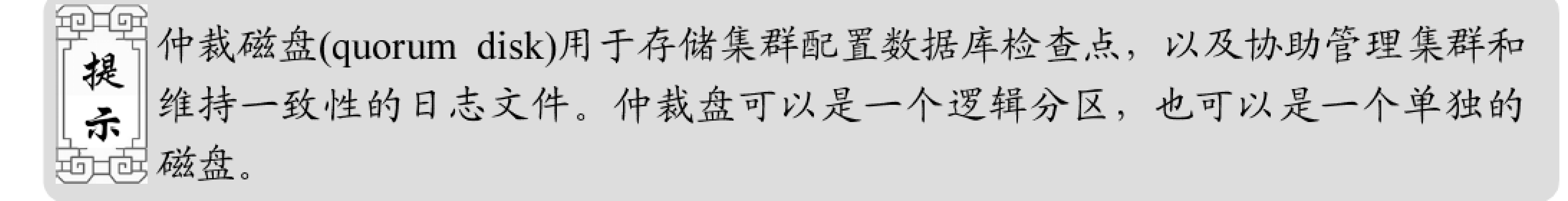
8】 输入在安装时创建的群集服务帐户的“用户名”和“密码”。

9】 群集配置完成，单击“完成”结束。

10】 群集配置完成后，选择磁盘阵列上的一个 LUN 为仲裁盘。

11】 完成节点 1 的配置后，在另一台机器上也安装群集服务，完成后打开群集管理器。

12】 当弹出群集连接向导时，选择“加入现有的群集”，根据向导完成节点 2 的配置。



仲裁磁盘(quorum disk)用于存储集群配置数据库检查点，以及协助管理集群和维持一致性的日志文件。仲裁盘可以是一个逻辑分区，也可以是一个单独的磁盘。

15.4.2 配置心跳网络

1】 启动“群集管理器”。
2】 在左窗格中，单击“群集配置”，再单击“网络”，右击用于专用网络(心跳检测专用)的适配器，然后选择“属性”命令。
3】 单击“仅用于内部群集通信(专用网络)”，如图 15.9 所示。
4】 单击“确定”按钮。
5】 右击用于公用网络的适配器，然后选择“属性”命令(如图 15.10 所示)。
6】 选中“针对群集应用启用该网络”复选框。
7】 选中“所有通信(混合网络)”单选按钮，然后单击“确定”按钮。

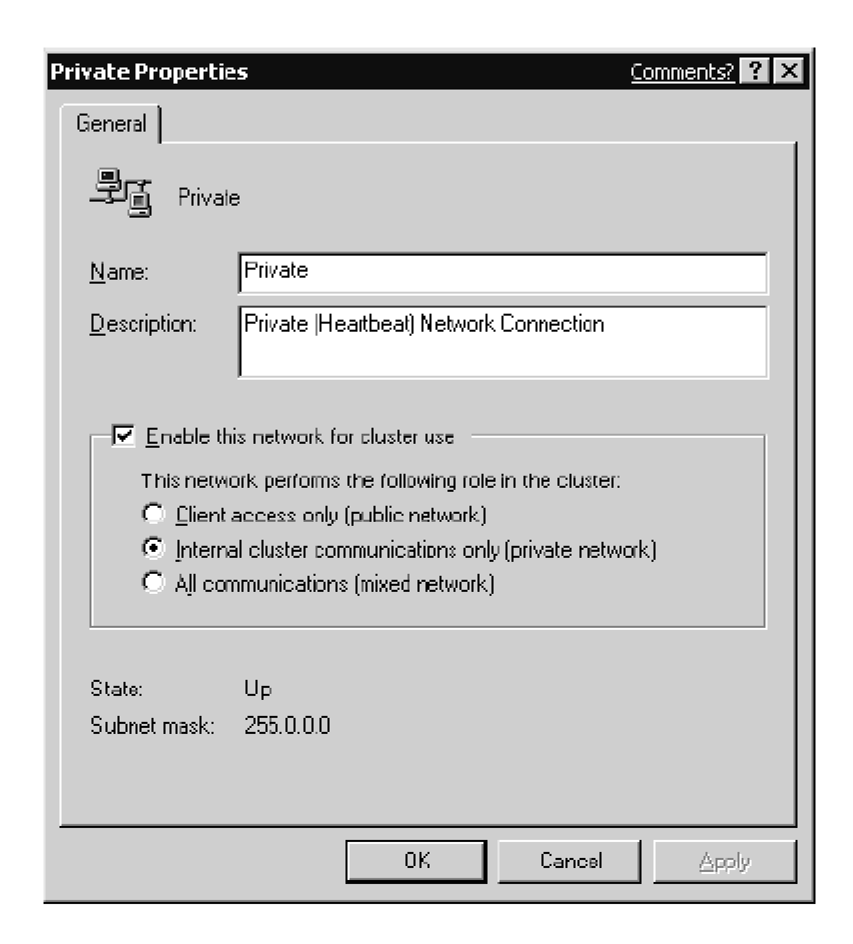


图15.9 配置专用网络

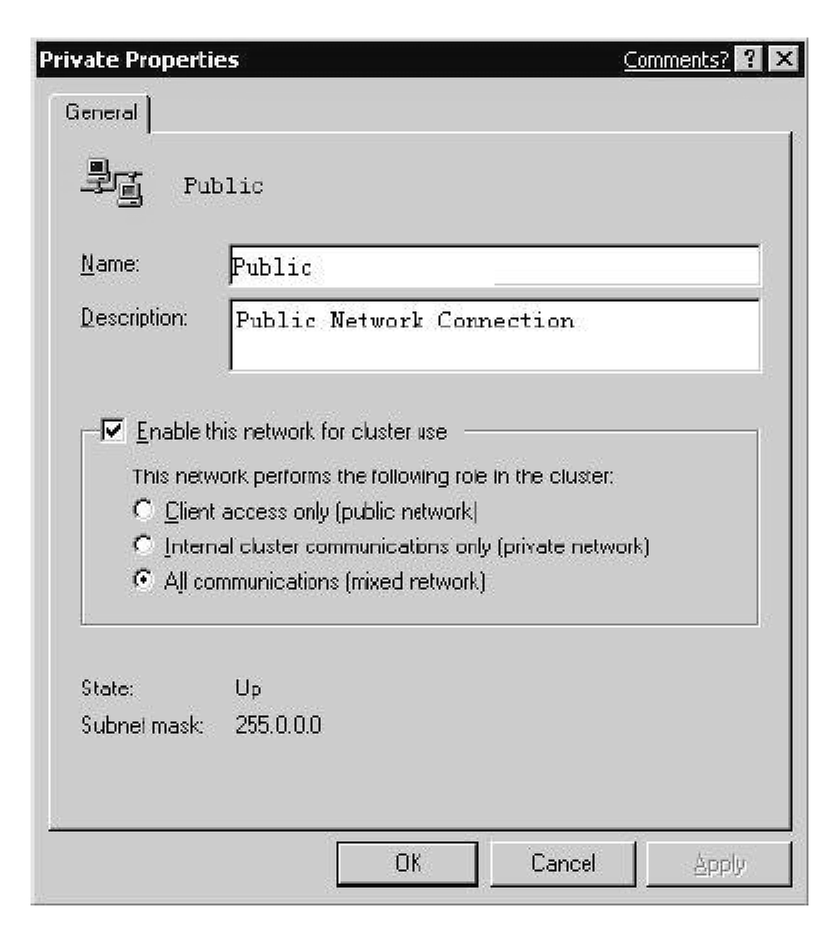


图15.10 配置公用网络

15.4.3 测试安装

在“安装”程序结束后，有几种验证群集服务安装的方法，其中包括如下。

1】 群集管理器：如果仅完成了节点 1 的安装，启动“群集管理器”，然后尝试连接到群集。如果已安装了第二个节点 2，可在任意一个节点上启动“群集管理器”，然后确认第二个群集显示在列表上。
2】 查看启动服务：使用管理工具中“服务”，确认群集服务已显示在列表上并已启动。
3】 事件日志：使用“事件查看器”检查系统日志中的“ClusSvc”条目。会看到有关确认群集服务已经顺利形成或加入一个群集的条目。
4】 群集服务注册表项：确认群集服务安装程序将正确的项写入注册表。可以在 HKEY_LOCAL_MACHINE\Cluster 下找到许多注册表设置。
5】 选择“开始”→“运行”菜单命令，然后在弹出的对话框中，输入“虚拟服务”名称。确认可以连接并看到资源。

15.4.4 测试故障转移

验证资源将执行故障转移。

选择“开始”→“程序”→“管理工具”菜单命令，然后单击“群集管理器”，如图 15.11 所示。

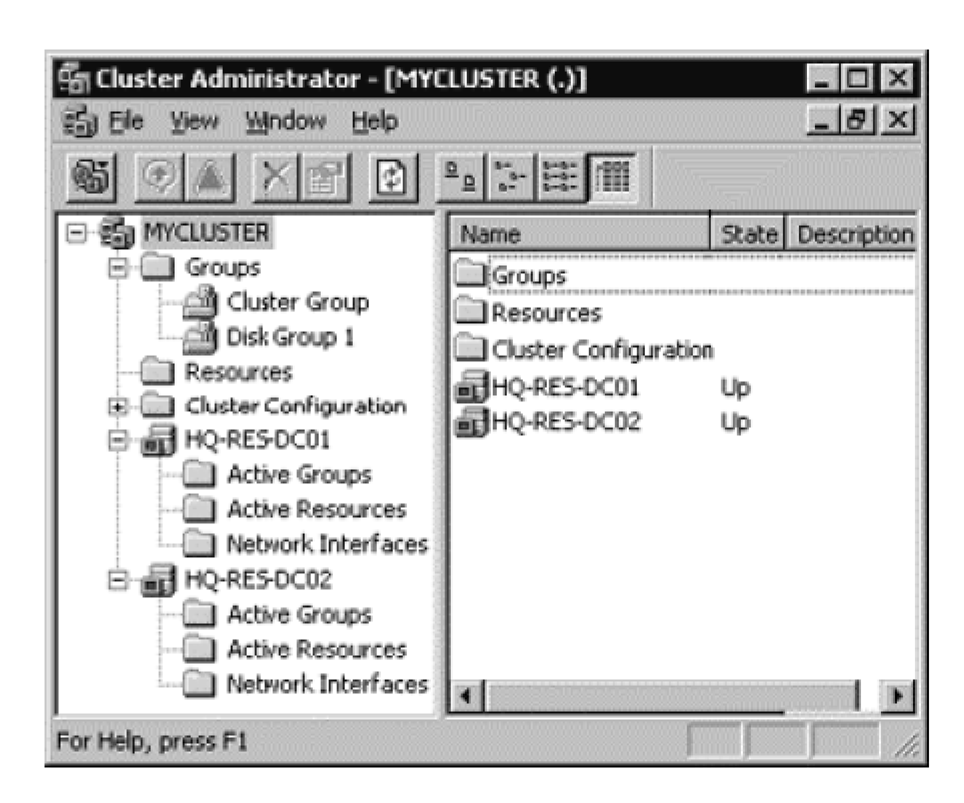


图 15.11 群集管理器主界面

右击“磁盘组 1”组，然后单击“移动组”。该组及其所有资源将转移到另一个节点。稍后，“磁盘 F:、G:”将在第二个节点上实现联机。在窗口中观察该转移。退出“群集管理器”。

15.5 实例：SQL Server 群集安装配置

上面我们已经设置好了 MSCS 群集基础平台，下面介绍在这个平台上安装 SQL Server 数据库。SQL Server 2000 的群集安装配置已经直接集成到了 SQL Server 2000 的数据库安装向导中，能够自动识别到 Windows Server 2003 上的群集系统并启用数据库虚拟服务器选项，实现 SQL Server 2000 群集虚拟服务器在两台服务器上的自动安装配置。安装完成后，须安装 SQL2000 SP3 补丁包。

确保 SQL Server 2000 群集在两台服务器上的自动安装配置，两台服务器 MS-Clus-01a 与 MS-Clus-01b，以及共享磁盘柜都须处于开机在线状态。

15.5.1 安装 SQL Server

1】 在接管了 SQL 数据盘(磁盘 Y:)的节点服务器 MS-Clus-01a 上，放入 SQL Server 2000 企业版安装光盘，启动 SQL Server 2000 的安装向导，如图 15.12 和图 15.13 所示。

2】 安装向导进入“计算机名”界面后，会自动识别到 Windows Server 2003 的群集系统，选择“虚拟服务器”选项，输入虚拟 SQL Server 名称“MS-Clus-SQL”，单击“下一步”按钮，如图 15.14 所示。



图15.12　共享磁盘 Y 被 MY-CLUS-01A 节点掌管



图15.13　安装 SQL Server



图15.14　输入虚拟 SQL Server 名称

3】 在“故障转移群集”对话框中输入 IP 地址“192.0.0.4”，选用网络 Public，单击“添加”按钮，使其添加到列表中，即这个 IP 地址属于公用网络。然后单击“下一步”按钮，如图 15.15 所示。

4】 在“群集磁盘选择”对话框中选择“组 0”的共享磁盘“Y：”，然后单击“下一步”按钮，如图 15.16 所示。



图15.15　配置虚拟服务器 IP 地址



图15.16　选择用于存放数据的磁盘

5】 在“群集管理”对话框，确保“MS-CLUS-01A”与“MS-CLUS-01B”都在“已配置节点”列表中，然后单击“下一步”按钮，如图 15.17 所示。

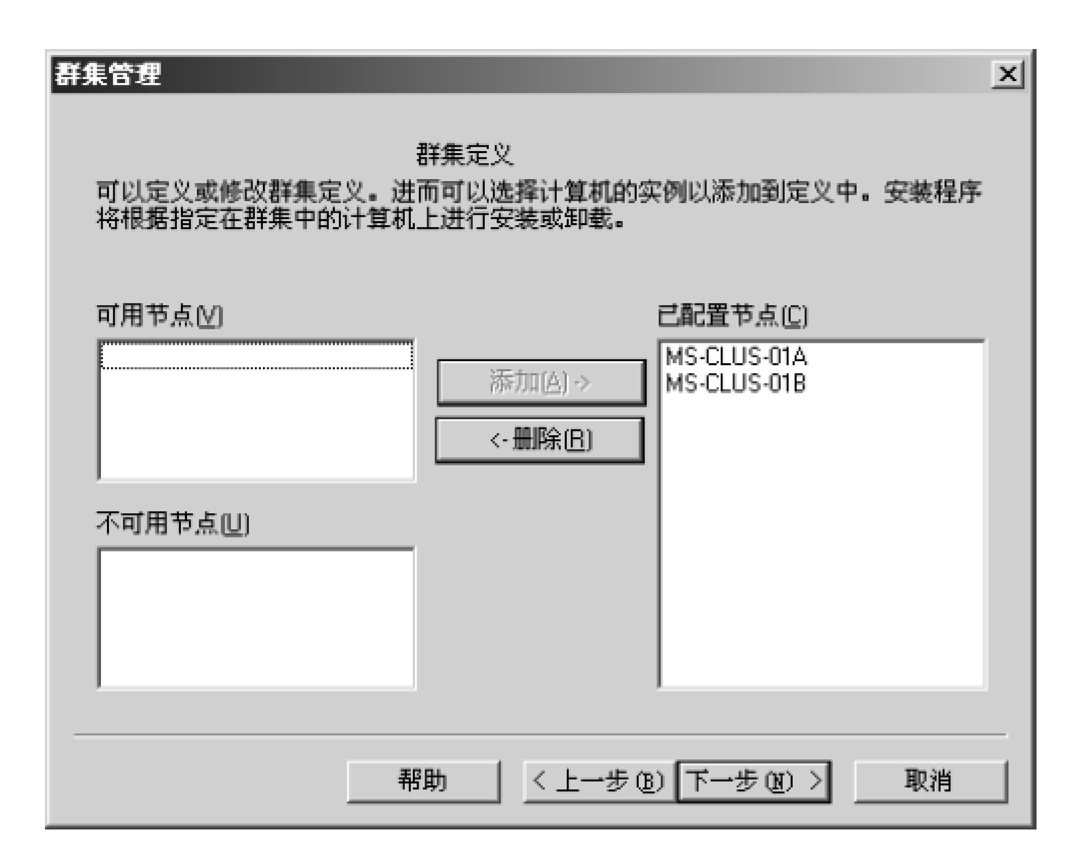


图15.17 选择群集中要使用到的节点

6】 在“远程信息”对话框，输入用户名、密码及域名，然后单击“下一步”按钮，如图 15.18 所示。

7】 在“实例名”对话框，选中“默认”复选框，然后单击“下一步”按钮，如图 15.19 所示。



图15.18 输入账户信息

图15.19 实例名窗口

8】 在“安装类型”对话框中选中“典型”单选按钮，由于前面磁盘选择了“组 0”的“磁盘 Y:”，“目的文件夹”的“数据文件”自动定位到 Y:盘，而 SQL 程序文件则会自动安装到 MS-Clus-01a 与 MS-Clus-01b 的本地盘相关目录下，单击“下一步”按钮，如图 15.20 所示。

9】 在“服务帐户”对话框，选中“对每个服务使用同一帐户”单选按钮，由于是群集配置“使用本地系统帐户”单选按钮为不可用，输入用户名、密码及域名，然后单击“下一步”按钮，如图 15.21 所示。

10】 在“身份验证模式”对话框中选中“混合模式”单选按钮，输入 sa 密码，然后单击“下一步”按钮，如图 15.22 所示。

11】 安装完成后，打开“群集管理器”，在“群集配置”下可看到“资源类型”中多了两个 SQL Server 的资源，这是因为 SQL Server 2000 企业版为 Cluster-Aware 的应用系统，安装配置时自动添加了支持 Cluster 的服务组件，如图 15.23 所示。



图15.20　安装目的选择



图15.21　输入账户信息

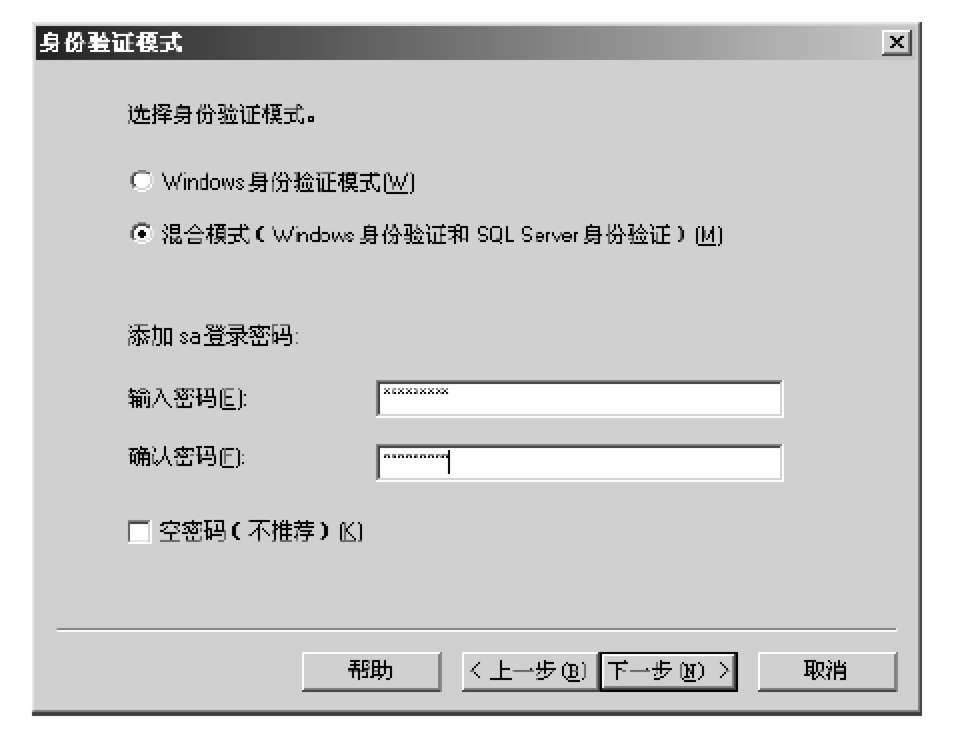


图15.22　身份验证配置



图15.23　新加入的资源(1)

12】 单击“组 0”，可看到除原有的“磁盘 Y:”外，新添了 5 个 SQL 资源，而且都已联机，说明 SQL Server 2000 群集安装配置完成，如图 15.24 所示。

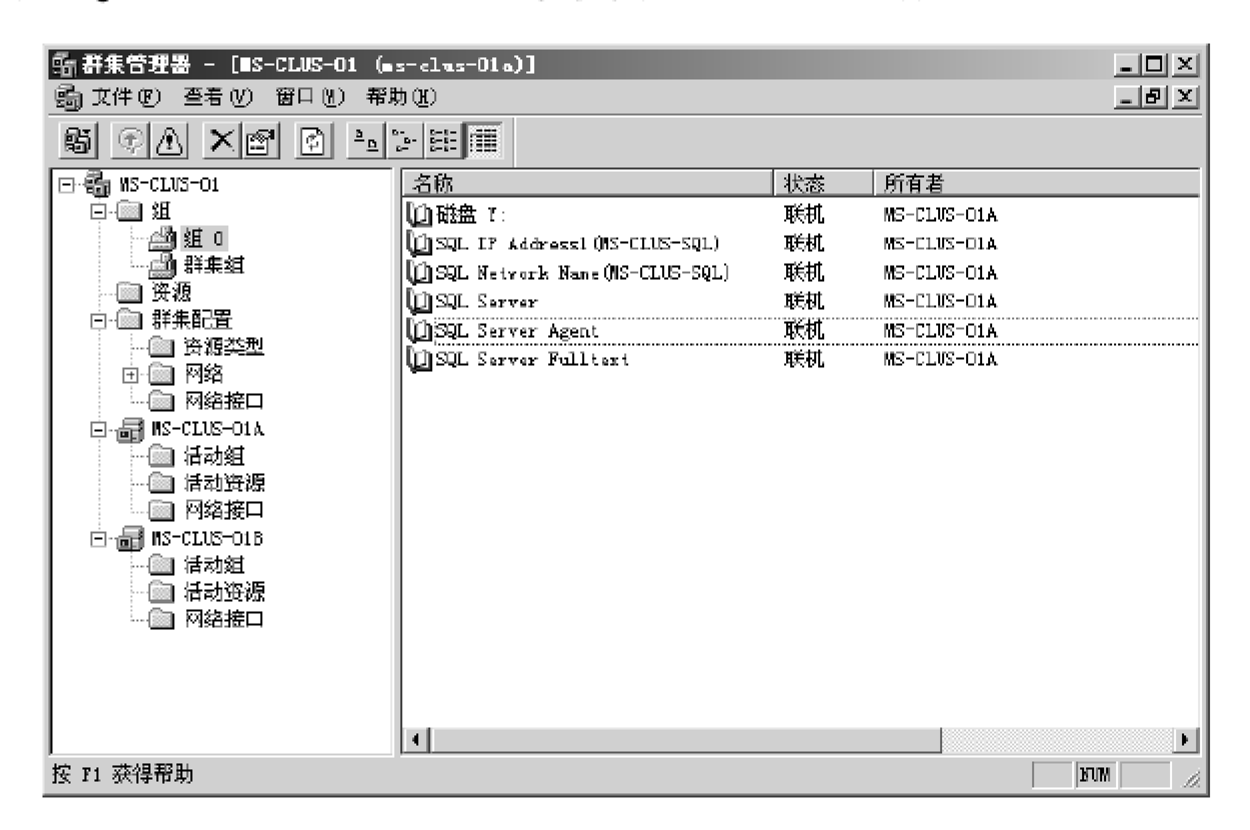


图15.24　新加入的资源(2)

15.5.2　验证 SQL 数据库群集功能

在一台客户机上安装 SQL Server 2000 的“企业管理器”与“查询分析器”，来测试验证数据库的 FailOver 功能。

1】 打开“企业管理器”，注册数据库“192.0.0.4”，即虚拟数据库的 IP 或服务器名，如图 15.25 所示。

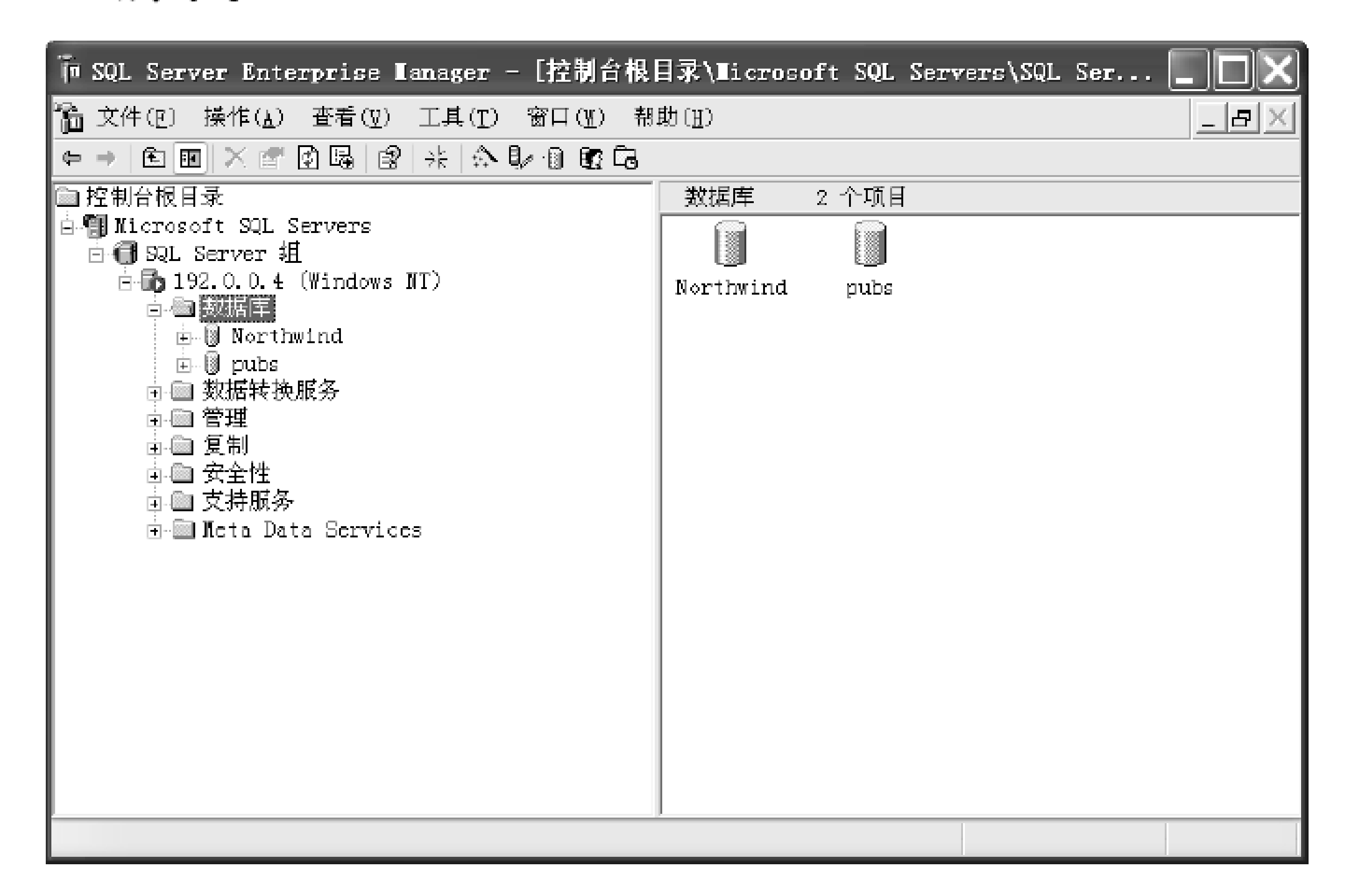


图15.25　连接到虚拟服务器

2】 新建一个测试数据库“MytestDB”，如图 15.26 所示。

3】 在数据库“MytestDB”中新建表“employee”，并添加几条记录，如图 15.27 所示。



图15.26　创建新数据库“MytestDB”

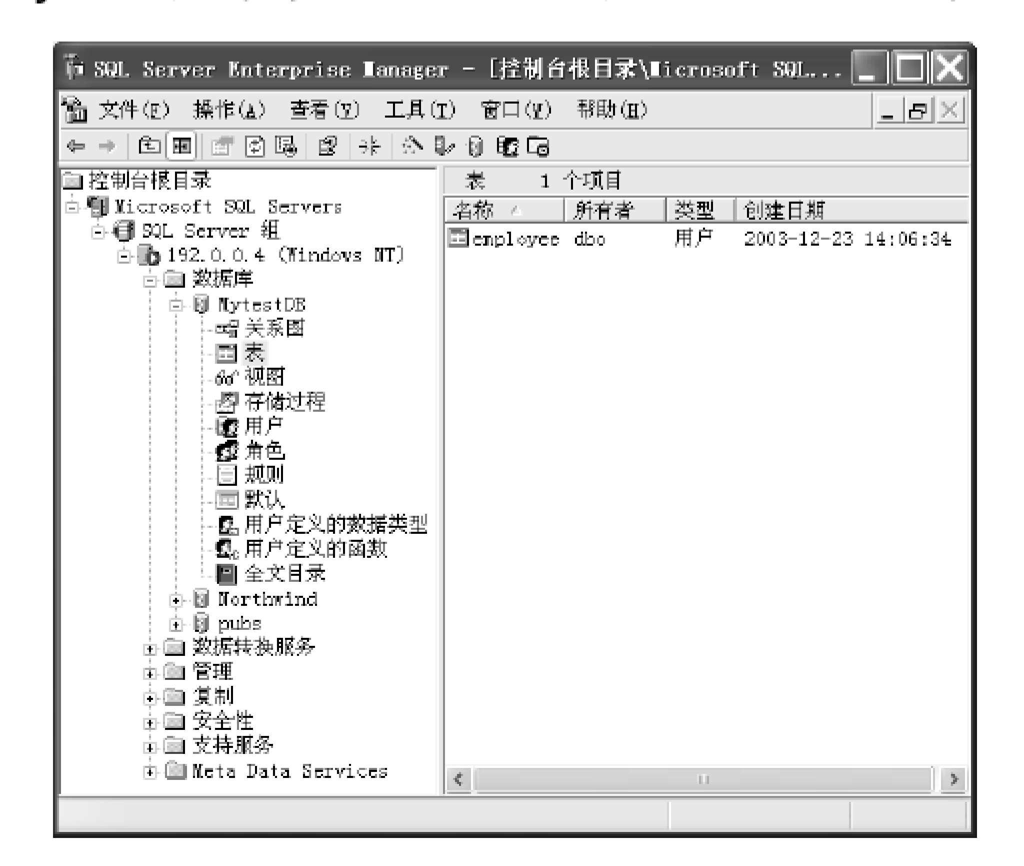


图15.27　新建表

4】 打开“查询分析器”，连接到 192.0.0.4 的数据库“MytestDB”，检索表“employee”返回数据，如图 15.28 所示。



图15.28　employee 返回数据

5】 在“群集管理器”中移动数据库资源组“组 0”(MS-Clus-01A→MS-Clus-01B)，进行资源切换，如图 15.29 和图 15.30 所示。

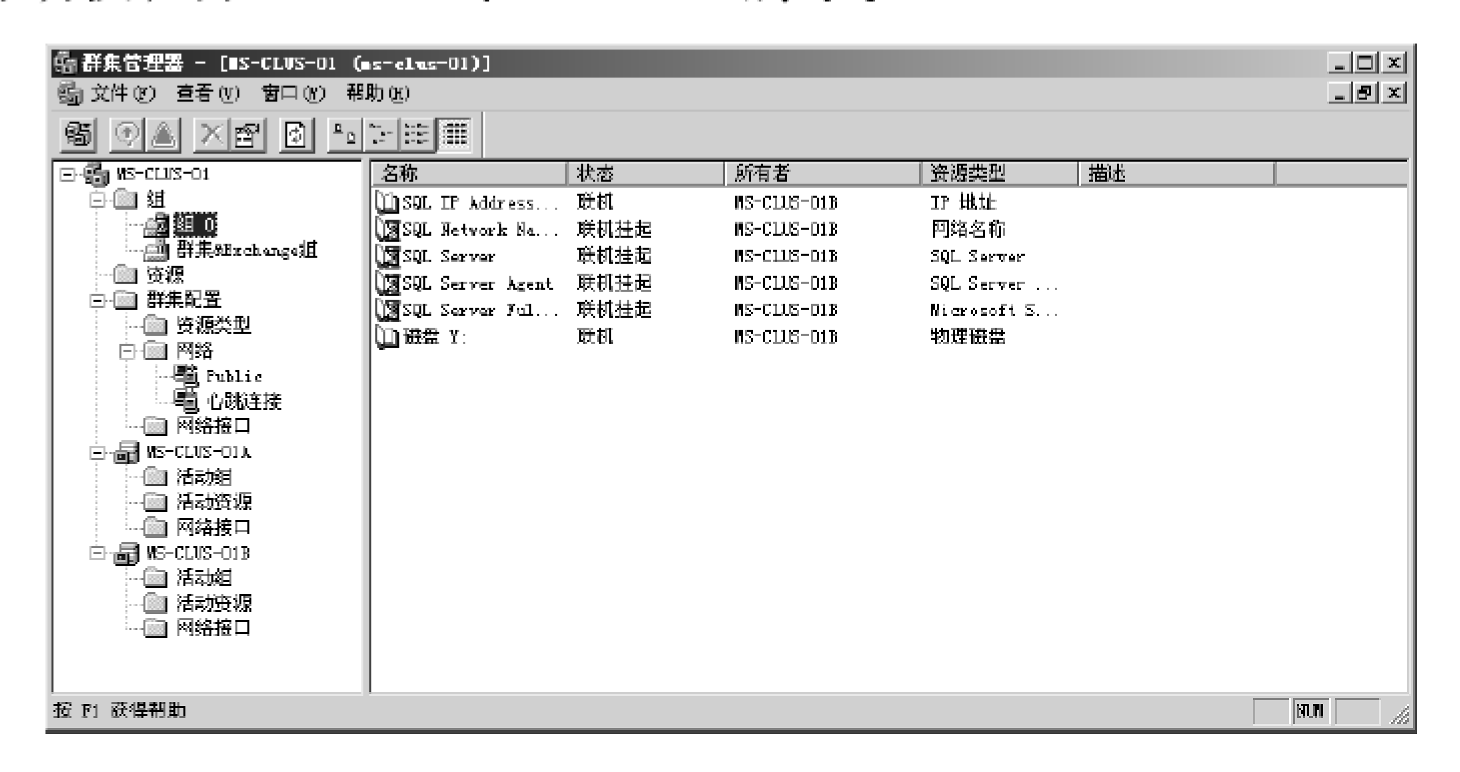


图15.29 手动切换资源(1)



图15.30 手动切换资源(2)

6】 移动“组 0”过程中，在“查询分析器”中持续执行数据检索，刚开始“连接中断”检索不到数据，几十秒钟后又恢复正常，能顺利检索到数据，如图 15.31 和图 15.32 所示。

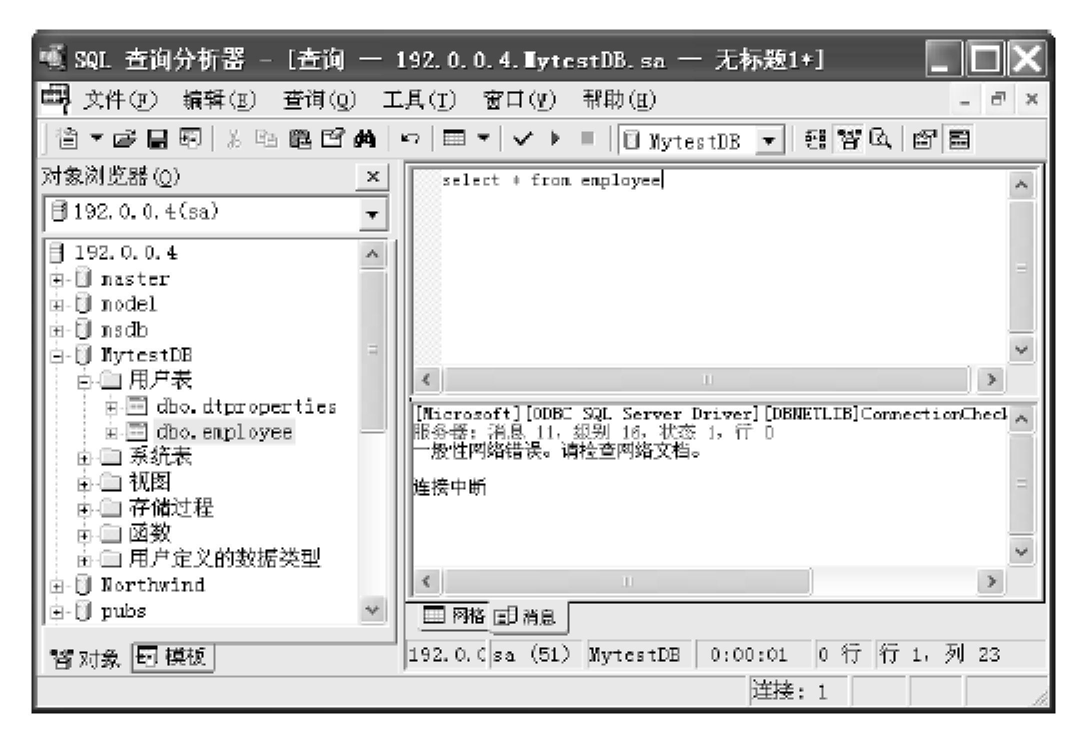


图15.31 服务中断



图15.32 服务恢复

测试验证表明，此配置方案可以实现数据库的 FailOver，而且切换时间在 30 秒左右。

15.6 小结：世界本身就是一个群集

世界就是一个群集。任何目前所理解的物质，都是由更小的“粒子”组成的群集。

或者说，任何“物质”，其实都是由世界本源的基本公式叠加而成的庞大公式。任何逻辑，都是这些“物质”即公式的叠加演算过程。逻辑是“物质”变化过程的体现。任何对逻辑的改变，最终体现到对“物质”的改变，而“物质”的改变，又体现为逻辑。

读 书 笔 记



数据保护和备份技术

- 数据保护
- 快照
- CDP
- 数据备份
- 备份通路

数据是表示信息的信息。数据丢失或者损坏，信息也就丢失或者被扭曲了。为了防止数据意外丢失或者损毁，人们想出了一系列的方法来保护当前的数据。此外，为了最大程度地保护数据，人们还在其他地方保存数据的一份或者多份副本。

数据保护和数据备份看似简单，实则很复杂。数据保护和备份已经成为存储领域的一门分支学科。本章就这个分支作简要介绍，并给出几个案例。

16.1 数 据 保 护

数据保护，就是对当前位置上的数据进行备份，以防突如其来的磁盘损坏，或者其他各种原因导致的数据不可被访问，或者部分数据损坏，影响到业务层。备份后的数据，可以在数据损毁之后恢复到生产磁盘上，从而最大程度的降低损失。

16.1.1 数据保护的方法

从底层来分，数据保护备份可以分为文件级的保护和块级的保护。

1. 文件级备份

文件级的备份，即备份软件只能感知到文件这一层，将磁盘上所有的文件，通过调用文件系统接口备份到另一个介质上。所以文件级备份软件，要么依靠操作系统提供的 API 来备份文件，要么本身具有文件系统的功能，可以识别文件系统元数据。

文件级备份软件的基本机制，就是将数据以文件的形式读出，然后再将读出的文件存储在另外一个介质上。这些文件在原来的介质上，存放可以是不连续的，各个不连续的块之间的链关系由文件系统来管理。而如果备份软件将这些文件备份到新的空白介质上，那么这些文件很大程度上是连续存放的，不管是备份到磁带还是磁盘上。

磁带不是块设备，由于机械的限制，在记录数据时，是流式连续的。磁带上的数据也需要组织，相对于磁盘文件系统，也有磁带文件系统，准确来说应该叫做磁带数据管理系统。因为对于磁带来说，它所记录的数据都是流式的、连续的。每个文件被看作一个流，流与流之间用一些特殊的数据间隔来分割，从而可以区分一个个的“文件”，其实就是一段段的二进制数据流。磁带备份文件的时候，会将磁盘上每个文件的属性信息，和实体文件数据一同备份下来，但是不会备份磁盘文件系统的描述信息，比如一个文件所占用的磁盘簇号链表等。因为利用磁带恢复数据的时候，软件会重构磁盘文件系统，并从磁带读出数据，向磁盘写入数据。

在 2003 年之前，很多人都用磁带随身听来欣赏音乐。而 2006 年之后，就很少看到带随身听的人，大都换成了 MP3，MP4。随身听用的是模拟磁带，也就是说它记录的是模拟信号。电流强，磁化的就强；电流弱，磁化的就弱。磁转成电的时候也一样，用这种磁信号强弱信息来表达声音振动的强弱信息，从而形成音乐。Mp3 则是利用数字信息来编码声音振动强弱和频率信息。虽然由模拟转向数字，需要数字采样转换，音乐的质量相对模拟信号来的差，算法也复杂，但是它具有极大的抗干扰能力，而且可以无缝的和计算机结合，形成能发声的计算机(多媒体计算机)。

录音带、录像带，都是模拟信号磁带。用于文件备份的磁带，当然是数字磁带，它记录的是磁性的极性，而不是被磁化的强弱，比如用 N 极来代表 1，用 S 极来代表 0。

如果备份软件将文件备份到磁盘介质或者任何其他的块介质上，那么这些文件就可以是不连续的。块设备可以跳跃式的记录数据，而一个完整数据链信息，由管理这种介质的文件系统来记录。磁盘读写速度比磁带要高的多。

近年来出现了 VTL，即 Virtual Tape Library(虚拟磁带库)，用磁盘来模拟磁带。这个概

念看似复杂，其实实现起来无非就是一个协议转换器，将磁盘逻辑与磁带逻辑相互映射融合，欺骗上位程序让其认为底层物理介质是磁带，然后再按照磁盘的记录方式读写数据，这就是虚拟化的表现。这种方法，提高了备份速度和灵活性，用处很大。

提示 数据保护并不是阳春白雪，常用的赛门铁克公司的 Ghost，就是一种文件备份软件。它将一个分区或者整块磁盘上的文件，与磁盘分区表、mbr 等信息一同备份，打包成一个大文件，系统故障的时候，就可以用软件来读取这个文件，向磁盘中做恢复。Ghost 支持多种文件系统，包括 Linux 的 EXT 文件系统。

Veritas，CA 等厂家都有自己的文件级备份软件解决方案。

2. 块级备份

所谓块级的备份，就是备份块设备上的每个块，不管这个块上有没有数据，或是这个块上的数据属于哪个文件。块级别的备份，不考虑也不用考虑文件系统层次的逻辑，原块设备有多少容量，就备份多少容量。在这里“块”的概念，对于磁盘来说就是扇区 Sector。块级的备份，是最底层的备份，它抛开了文件系统，直接对磁盘扇区进行读取，并将读取到的扇区写入新的磁盘对应的扇区。

这种方式的一个典型实例，就是磁盘镜像。而磁盘镜像最简单的实现方式就是 RAID 1。RAID 1 系统将对一块(或多块)磁盘的写入，完全复制到另一块(或多块)磁盘，两块磁盘内容完全相同。有些数据恢复公司的一些专用设备“磁盘复制机”也是直接读取磁盘扇区，然后复制到新的磁盘。

基于块的备份软件，不经过操作系统的文件系统接口，通过磁盘控制器驱动接口，直接读取磁盘，所以相对文件级的备份来说，速度加快很多。但是基于块的备份软件备份的数量相对文件级备份要多，会备份许多僵尸扇区，而且备份之后，原来不连续的文件，备份之后还是不连续，有很多碎片。文件级的备份，会将原来不连续存放的文件，备份成连续存放的文件，恢复的时候，也会在原来的磁盘上连续写入，所以很少造成碎片。有很多系统管理员，都会定时将系统备份并重新导入一次，就是为了剔除磁盘碎片，其实这么做的效果和磁盘碎片整理程序效果一样，但是速度却比后者快的多。

16.2 高级数据保护方法

16.2.1 远程文件复制

远程文件复制方案，是把需要备份的文件，通过网络传输到异地容灾站点。典型的代表是 rsync 异步远程文件同步软件。这是一个运行在 Linux 下的文件远程同步软件。它可以监视文件系统的动作，将文件的变化，通过网络同步到异地的站点。它可以只复制一个文件中变化过的内容，而不必整个文件都复制，这在同步大文件的时候非常管用。

其实 FTP 工具也是一个很好的远程文件复制工具，只不过不能做到更加灵活和强大而已。

16.2.2 远程磁盘(卷)镜像

这是基于块的远程备份。即通过网络将备份的块数据传输到异地站点。远程镜像(远程实时复制)又可以分为同步复制和异步复制。同步复制，即主站点接受的上层 IO 写入数据，必须等这份数据成功的复制传输到异地站点，并写入成功之后，才通报上层 IO 成功消息。异步复制，就是上层 IO 主站点写入成功，即向上层通报成功，然后在后台将数据通过网络传输到异地。前者能保证两地数据的一致性，但是对上层响应较慢。而后者不能实时保证两地数据的一致性，但是对上层响应很快。

所有基于块的备份措施，一般都是在底层设备上进行，而不耗费主机资源。

现在几乎各个盘阵厂家的中高端产品，都提供远程镜像服务，比如 IBM 的 PPRC，EMC 的 SRDF，HDS 的 Truecopy，Netapp 的 SnapMirror 等。

16.2.3 块(快)照数据保护

远程镜像，或者本地镜像，确实是对生产卷数据的一种很好的保护，一旦生产卷故障，可以立即切换到镜像卷。但是这个镜像卷，一定要保持一直在线状态，主卷有写 IO 操作，那么镜像卷也有写 IO 操作。如果某时刻想对整个镜像卷进行备份，需要停止读写主卷的应用，使应用不再对卷产生 IO 操作，然后将两个卷的镜像关系分离，这就是拆分镜像。

拆分之后，可以恢复上层的 IO。由于拆分之后已经切离的镜像关系，所以镜像卷不会有 IO 操作。此时的镜像卷，就是主机停止 IO 那一刻的原卷数据的完整镜像，此时可以用备份软件，将镜像卷上的数据，备份到其他介质。

拆分镜像，是为了让镜像卷保持拆分一瞬间的状态，而不再继续被写入数据。而拆分之后，主卷所做的所有写 IO 动作，会以 bitmap 的方式记录下来。bitmap 就是一份位图文件，文件中每个位都表示卷上的一个块(扇区，或者由多个扇区组成的逻辑块)，如果这个块在拆分镜像之后，被写入了数据，则程序就将 bitmap 文件中对应的位从 0 变成 1。待备份完成之后，可以将镜像关系恢复，此时主卷和镜像卷上的数据是不一致的，需要重新做同步。程序搜索 bitmap 中所有为 1 的位，对应到卷上的块，然后将这些块上的数据，同步到镜像卷，从而恢复实时镜像关系。

可以看到，以上的过程是十分复杂繁琐的，而且需要占用一块和主卷相同容量大小的卷作为镜像卷。最为关键的是，这种备份方式需要停掉主机 IO，这对应用会产生影响。而“快照技术”解决了这个难题。快照的基本思想是，抓取某一时间点磁盘(卷)上的所有数据，而且完成速度非常快，就像照相机快门一样。

下面介绍快照的底层原理。

1. 基于文件系统的快照

我们知道，文件系统管理思想的精髓就在于它的链表、B 树和位图等结构，也就是元数据(metadata)，以及对这些元数据的管理方式。文件系统其实对底层磁盘是有点“恐惧”的，我们可以计算一下，一个 100GB 的磁盘上，有超过 2 亿个扇区。文件系统是如何管理这 2

亿个扇区，又如何知道某个扇区正在使用呢？如果使用的话，是分配给哪个文件或者文件的一部分呢？

文件系统首先将扇区组合成更大的逻辑块来降低管理规模。NTFS 最大每个块可以 4KB，也就是 8 个扇区一组形成一个簇(块，block)。这样，2 亿的管理规模便会除 8，缩小到 1.2 千万的管理规模，虽然存储空间可能有所浪费，但是切实降低了管理成本。

其次，文件系统会创建所管理存储空间上所有簇(块)的位图文件，这个文件有固定的入口，文件系统能在 1.2 千万个块中快速定位到这个文件入口并读写。位图文件中每个位代表卷上的一个簇(或者物理扇区，视设计不同而决定)，如果簇正在被某个文件使用，这个簇在位图中对应的位的值就为 1，否则为 0。

再次，文件系统还保存一份文件和其所对应簇号的映射链，这个映射链本身以及簇位图本身也是文件，也要有自己的映射链，所以针对这些重要的元数据，必须有一个固定的入口，用来让文件系统程序读入并且遍历所有文件系统元数据。通常将这个初始固定地址入口称为 root inode。

当向卷中写入一个新文件，文件系统首先会查找簇位图，找到位值为 0 所对应的簇号，并计算所需的空间，然后分配这些簇号给这个文件。它首先将文件实体数据写入对应的簇，然后再去文件－簇号映射图中更新，将新文件与其对应的簇映射关系记录下来，最后到簇位图中将这些簇对应的位的值从 0 改为 1。如果要删除这个文件，则直接在文件的簇号映射链中抹掉对应的文件极簇的记录，然后在簇位图中将对应簇的位值从 1 改为 0。

文件系统并不会抹掉这个被删除文件所对应卷上的簇中的实际数据，如果用扇区读写软件来提取这个簇，就会得到这个文件的部分内容。虽然这些簇中依然有内容，但是对于文件系统来说，这些簇是可重用的，一旦有新文件写入，新文件的数据便会覆盖原来簇中的数据。

所以，对于一个文件系统来说，最重要的不是卷簇中的数据，而是文件－簇号映射链和位图等这些元数据。比如，想要破坏某个文件系统中的某个文件，我们不必费劲地修改某个文件对应簇中的实际内容，只要修改一下文件－簇号映射链中关于这个文件所对应的实际簇号的记录即可，让它指向其他簇号。

这样，文件读出来的内容就不是原有内容。此外，如果修改了文件系统中对应簇中的数据，文件系统也根本感知不到这些动作，因为它所查询的是文件－簇号映射链，它只知道某个文件对应着哪些簇，而不关心这些簇是否被改过，它想关心也无法关心。

正因为文件系统只是根据它所记录的这些映射图表来管理文件，所以才使得快照成为可能。为什么呢？



如果我们想抓取某个卷在某一时刻下的全部数据，就可以像照相机一样，对一个正在移动的物体实施拍照，某一时刻，快门打开，曝光后关闭。照相机可以保存在底片上，但照下的数据如何保存呢？

如果一个卷上有 100GB 容量的文件，照下来之后，用另一块磁盘将数据全部拷贝出来，至少需要 20 分钟的时间，且在这段时间内，不允许再有任何数据被写入这个卷，因为我们

要抓取的是这个卷在这个时刻瞬间的数据。

对于拍照移动中的物体，要想保证照片清晰没有重影，就要降低曝光时间，但是曝光时间越短，照片感光度就会越低，也会影响质量。同样，对于一个正在进行频繁 IO 写入的数据卷，要照下它某时刻的样子，也需要减少“曝光”时间。但是刚才已经说过，100GB 的数据，拷贝出来需要 20 分钟时间，这个“曝光”时间太长了。要么物体在拍摄的时候保持静止(拷贝卷的时候停止 IO)，要么就减少曝光时间(加快拷贝速度)，这两样，我们一样也保证不了，因为停止 IO 会直接影响上层应用，而且也无法在几秒钟(应用停止 IO 可接受的时间内)之内将 100GB 的数据拷贝出来。难道就真的没有办法了么？非也。



如果我们只花几秒甚至 1 秒钟的时间，把某时刻的文件系统中的映射图表，以及下级链表等元数据复制出来并保存，那么是不是就可以说，我们照下了这个时间点处卷上的所有数据呢？

绝对是的。因为文件系统只根据这些映射图来对应卷上的实际数据，所以只要有了映射图，就可以按图索骥，找到并拥有实际的数据。但是，将此时刻的映射图拷贝出来，我们也只得到了一个图纸而已，因为实际数据在卷上，是实实在在的物理卷，而不是在图纸上，所以我们必须保证卷上的数据不被 IO 写入，而同时又不能影响应用，既然不能影响应用，就要让 IO 继续执行，这简直是个大大的矛盾啊！



原块不能被写入新数据，这个绝对要保证，否则这张“照片”就花脸了，而同时应用的 IO 也必须要持续不断的执行(写入)，既然原块不让写入了，那能不能写入到其他空闲的地方呢？

当然可以！每当遇到需要向原卷写入新文件或者更新旧文件，文件系统便将这些更新数据写入一个新的空闲的地方去，然后在它的映射图中加入对应的条目。由于我们已经保存了原来的映射图，而且也拥有一份不让写入的原卷实体数据，所以不怕它修改。文件系统当前的映射图(元数据)始终描述的是当前的映射关系。这样，应用继续执行 IO，同时，我们也可以将照下来的原卷数据拷贝出去，从而得到一份某时刻的瞬间的卷数据。我们完全可以选择不拷贝这份快照，就让他留在那，因为快照根本不占用额外的空间，除非针对这个卷有新的 IO 写入，则新簇会写向其他地方，从而占用相应大小的空间，一旦原卷所有簇都被更新了，那么也就意味着在空闲空间内逐渐生成了一个完整的新卷了，其占用与原卷相同大小的空间。

图 16.1 所示为快照的示意图。当前的文件系统指针图和快照中的指针图，在原块没有被覆盖的时候，是指向同一位置的，一旦原块有写 IO 执行，则这个块会被写到其他空闲块或者其他卷，而将当前活动文件系统对应指针指向这个新块，其他指针不变。后面我们会讲到两种管理快照块的方式。

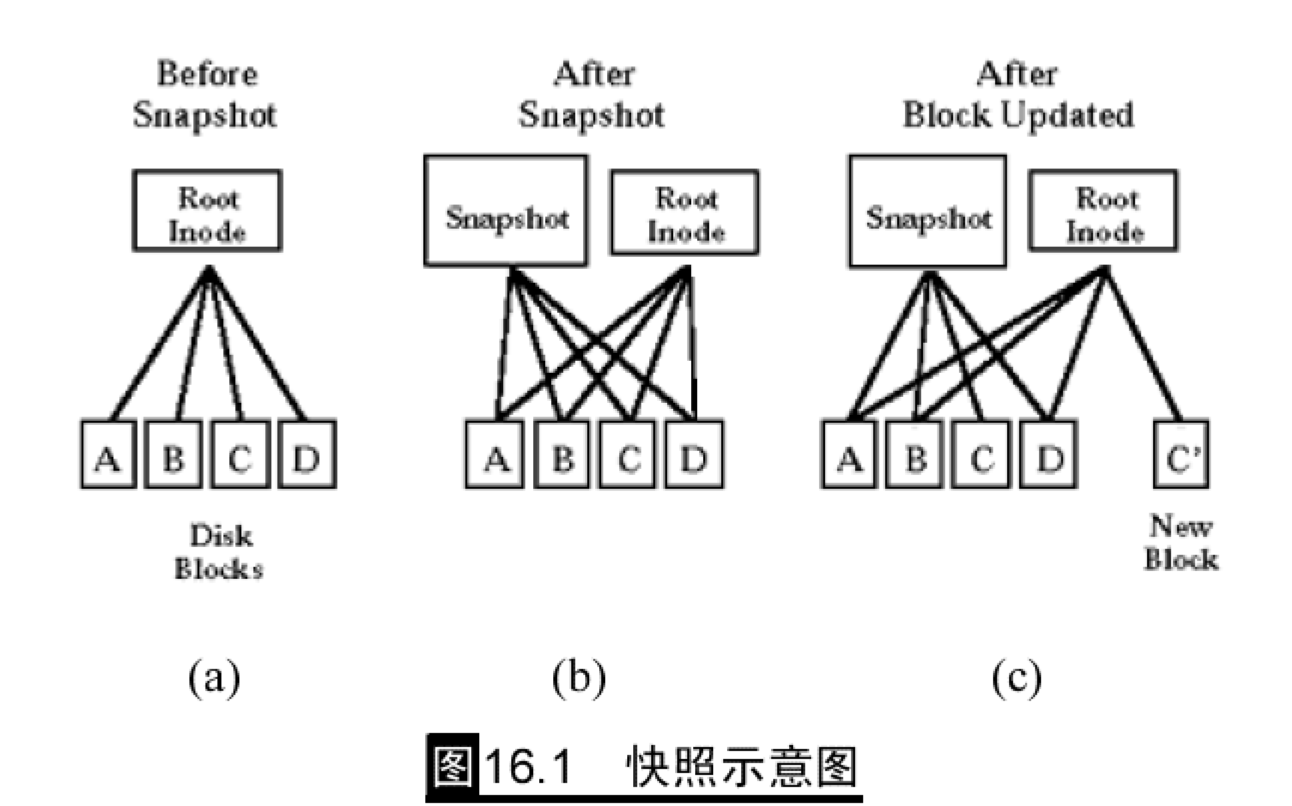


图16.1 快照示意图

> 提示
> NetApp 公司的 WAFL 文件系统利用的快照方式，是很有特色的。每次快照，它只将根 inode 拷贝并保存，而不保存下级链表 inode。之所以敢这么做的原因，是因为 WAFL 从来不会覆盖写入某个文件对应的旧块。不管是元数据还是实体数据，WAFL 统统写入到卷的空闲块上(根节点 inode 映射图位置恒定，每次更新会覆盖写)。这样，在只复制了跟节点 inode 之后，由于下级链表 inode 均不会被覆写，所以同样可以保存瞬间的 snapshot。其他的快照实现方式，一般都将所有 inode 复制并保存，因为它们的 inode 都是恒定位置的，只能全部覆写。

2. 基于物理卷的快照

基于物理卷的快照，相当于给物理卷增加了一个“卷扇区映射管理系统”。我们知道，卷扇区应当是由文件系统来组织和管理的，但是为了减轻文件系统负担，人们在底层卷这个层次实现快照。卷扇区都是用 LBA 来编号的，实现快照的时候，程序首先保留一张初始 LBA 表，每当有新的写入请求的时候，程序将这些请求的数据，写入另一个地方(一般是一个新卷，专为快照保留的)，并在初始 LBA 表中做好记录，比如：

原始 LBA：卷 A 的 10000 号，映射到 LBA：卷 B 的 100 号

以上映射条目的产生，是由于有 IO 请求写入数据到卷 A 的 10000 号 LBA，由于做了快照，卷 A 在这个快照被删除之前不允许写入，所以将这个写入请求的数据，重定向写到卷 B 的 100 号 LBA 扇区上。值得说明的是，文件系统不会感知到这个重定向动作，FS 在它的映射图中依然记录了卷 A 的 10000 号 LBA 地址而根本不知道还有个卷 B。

此时，如果文件系统生成了一个 IO 请求读取，或者写入卷 A 的 10000 号 LBA 扇区，那么运行在卷层的快照程序便会查找快照重定向映射表，发现卷 A 的 10000 号 LBA 其实已经被重定向到了卷 B 的 100 号，然后读取或者写入卷 B 的 100 号扇区。由于每次 IO，程序均会查找这份快照映射表，所以增加了处理时间，降低了一些性能。这种方式称为 Write Redirect，意思是重定向写，也就是将更新的数据写入另一个地方，原卷数据丝毫不动，用指针来记录这些重定向的地址。在利用 Write Redirect 方式做了快照之后，针对随后的每个针对这个卷的上层 IO，程序都需要查表确认是否需要重定向到新卷（或者本卷为快照所保留的空间），这种做法对性能是有较大影响的，为此，有人发明了另一种方式来保存快照数据。

快照生成之后，如果上层有针对原卷某个或者某些自从快照之后从来未被更新过的 LBA

块的写 IO 请求，则在更新这些 LBA 扇区之前，先将原来扇区的内容拷贝出来，放入一个空闲卷，然后再将新数据写入原卷。也就是说，旧数据先占着位置，等什么时候新数据来了，旧数据再让位，一旦原卷某个 LBA 的块在快照之后被更新过了，则以后再针对这个 LBA 块的写 IO，可以直接覆盖，不需要提前拷贝，因为第一次更新此块的时候已经将原块数据拷贝保留了。这样，原卷上的数据随时都是当前最新的状态，所以针对快照之后的每个上层 IO，不必再遍历映射表，直接写向原卷对应的地址，如果是写入一个快照之后从未被更新过的块，则需提前将原块拷贝保留，这种方式称为“Copy On Write”，写前拷贝。

在“照”下了这一时刻卷上的数据之后，为了保险起见，最好对那个时刻的数据做一个备份，也就是将快照对应的数据复制到另外的磁盘或者磁带中。如果不备份快照，那么一旦卷数据有所损毁，快照的数据也不复存在，因为快照与当前数据是共享 LBA 扇区的(如果没有更新原卷扇区的话)。

3. Write Rediret 方式和 Copy On Write 方式比较

不管是 Copy On Write 还是 Write Redirect，只要上层向一个在快照之后从来没被更新过的扇区写 IO，这个 IO 块就要占用新卷上的一个块(因为要保留原块的内容，不能被覆盖)，如果上层将原卷上的所有扇区块都更新了，那么新卷的容量就需要和原卷的数据量同样大才可以。但是通常应用不会写覆盖面百分之百，做快照的时候，新卷的容量一般设置成原卷容量的 30%就可以。

Copy On Write 方式下，快照完成之后，如果需要更新一个从来没有被更新过的块，则程序首先将这个块读出，再将其写入到新卷，然后将更新的数据覆盖写入到原卷对应的块，需要三步动作，一次读和两次写。Write Redirect 方式下，同样的过程只需要一次写入即可，即将更新数据直接写入到新卷，同时更新映射图中的指针(内存中进行)。所以 Write Redirect 相对 Copy On Write 方式在 IO 延迟上有优势。但是 Write Redirect 方式下，对于每个上层 IO，都必须遍历一下映射表，以便确定此 IO 请求的 LBA 地址是否已被重定向到新卷，而 Copy On Write 方式却不需要遍历这张表，因为当前最新的数据总是在原卷上。在卷级快照的情况下，Write Redirect 方式会有劣势。在文件系统级快照情况下，Copy On Write 方式和 Write Redirect 方式同样都要针对每个 IO 遍历当前文件系统中的元数据，这部分开销是一样的，而 Write Redirec 耗费 IO 方面的开销，就要比 Copy On Write 小得多了。

总之，卷级的快照，仿佛就是增加了一个“卷块映射系统”，其作用与文件系统大同小异，只不过文件系统处理的是文件名和块的映射关系，而“卷块映射系统”处理的是块与块的映射关系。

4. 快照的意义



快照所冻结下来的卷数据，无异于一次意外掉电之后卷上的数据。为什么这么说呢?

我们可以比较一下，意外断电同样是保持了断电所处时间点上的卷数据状态。

我们知道，不管是上层应用，还是文件系统，都有自己的缓存，文件系统缓存的是文件系统元数据和文件实体数据。并不是每次数据的交互，都同步保存在磁盘上，它们可以

暂时保存在内存中，然后每隔一段时间(Linux 系统通常为 30 秒)，批量 Flush 到磁盘上。当然编程的时候也可以将每次对内存的写，都 Flush 到磁盘，但是这样做效率和速度打了折扣。而且当 Flush 到磁盘的时候，并不是只做一次 IO，在数据量大时会对磁盘做多次 IO。如果快照生成的时间恰恰在这连续的 IO 之间生成，那么此时卷上的数据，实际上有可能不一致。

磁盘 IO 是原子操作(Atomic operation)，而上层的一次事务性操作，可以对应底层的多次原子操作。这其中的一次原子操作，没有业务意义，只有上层的一次完整的事务操作，才有意义。所以如果恰好在一个事务操作对应的多个原子操作的中间，生成块照，那么此时的快照数据，就是不完整的，不一致的。

文件系统的机制总是先写入文件的实体数据到磁盘，文件的元数据暂不写到磁盘，而是先保存于缓存中。这种机制是考虑到一些意外事件，如果 FS 先把元数据写入磁盘，而在准备写入文件实体数据的时候，突然断电了，那么此时磁盘上的数据是这么一个状态：FS 元数据中有这个文件的信息，但是实体数据并没有被写入对应的扇区，那么这些对应的僵尸扇区上原来的数据，便会被认为就是这个文件的数据，这显然后果不堪设想。

所以 FS 一定是先写入文件实体数据，完成之后再批量将元数据从缓存中 Flush 到磁盘，如果在实体数据写入磁盘，而元数据还没有写入磁盘之前断电，那么虽然此时文件实体数据在磁盘上，但是元数据没有在磁盘上，也就是说虽然有你这个人存在，但是你没有身份证，那么你就不能公开的进行社会活动，因为你不是这个国家的公民。虽然文件系统这么做，会丢失数据，但是总比向应用提交一份驴唇不对马嘴的数据强！

提示

实验：就拿 Windows 来说，首先创建一个文件，并在创建好的瞬间，立即断电，重启之后，会发现刚才创建的文件没了，或者复制一个文件，完成后立即断电，重启之后也会发现，复制的文件不见了，为什么？明明创建好的文件，复制好的文件，为什么断电重启就没了呢？原因很简单，因为断电的时候，FS 还没有把元数据 Flush 到磁盘上，你就给断电了，此时文件实体收据虽然还在，但是元数据中没有，那么当然看不到它了。

总之，快照极有可能生成一份存在不一致的卷数据。既然这样为何还要使用快照呢？因为相对于停机备份，人们更接受使用快照来备份数据，即使快照可能带来数据不一致。但停机备份所带来的损失，对于某些关键应用来说是不可估量的，而快照只需要几秒钟即可完成，应用只需静默几秒钟的时间。文件系统或者卷擅自将 IO 先存放到队列中，等待快照完成后，再继续执行。然后可以随时将这份快照对应的数据拷贝出来，形成备份。

使用快照必须承担数据不一致的风险。也可以这么形容：快照可以让你不用“意外磁盘掉电”，就能获得一个时间点瞬间磁盘备份，虽然数据此时是不一致的。快照可以任意生成，而占用的空间又不会很大(随原卷数据改动多少而定)，最重要的是，利用快照可以做在线快速恢复，只要快照没删除，恢复也同样仅仅需要几秒钟时间，与快照生成的道理一样，不用停机。因为利用快照恢复数据的时候，只要在内存中作一下 IO 重定向，那么上层 IO 访问的，就立即变成了以前时间点的数据了。这就是快照最大的意义。

快照可能不一致这个问题，也不是不能解决。既然快照无异于一次磁盘掉电，那么，利用快照恢复数据之后，文件系统可以进行一致性检查从而纠正错误。数据库管理系统也

同样会利用其日志来使得数据文件最终出于一致状态。

另外，有些快照的解决方案，是在主机上安装一个代理软件，当在存储设备上执行快照之前，代理软件会通知应用或者文件系统将缓存中的数据全部 Flush 到磁盘，然后立即生成快照，这样快照的一致性就得到了保护。

对于文件系统级的快照，应用程序可以直接在操作系统中看到快照，可以直接读取这份快照中的文件。而对于卷级的快照，只有存储设备自己可以管理，操作系统对此一无所知。

对卷做快照之后，一旦原先的数据块被更改/覆盖，则这些块要么在写入前被拷贝出来，要么原地锁住不动，新数据重定向写到空闲区域。总之，这两种情况下，系统内整体空余空间都要被额外数据占用而减小。

然而，如果对某个块进行二次更改/覆盖，则可以直接在这些新块之上进行操作而不需要写前拷贝或者再次重定向写入，除非在第一次更改/覆盖和第二次之间又创建了一个或者多个快照。



如果将原来的数据块删除了，在已经做了快照的情况下，系统将作出什么行为？删除原块对应的行为其实是在文件系统的 inode tree 中将对应的指针消除，表现为更改/覆盖对应 inode 所在的块，所以不管删除多大的文件，也只有对应的 inode 块被更改/覆盖，被删除数据依然存在于磁盘上，然而这些数据对应的块在当前活动文件系统下的簇位图中已经被标记为空闲块了，此时一旦又有新数据被追加写入到这些空闲块，那么依然会被重定向或者写前拷贝。

使用快照功能之前必须首先评估数据源 IO 行为，随着覆盖写入几率的增加，系统中可使用的空闲空间的数量也要随之增加。实际情况中，文件被覆盖写入的几率一般不高，比较高的是 Create+Write，Delete，Rename，Open，Read，Truncate，Append 等操作，其中 Delete，Rename，Truncate 操作会导致 Snapshot 占用额外空间，而这其中 Delete 和 Truncate 会导致被删除文件本身与其 inode 的 block 都占用额外空间，Rename 只会导致对应 inode block 占用额外空间。Create+Write 操作如果一旦覆盖了旧块，那么同样也会占用额外空间。



目前几乎所有厂商的存储产品都可以实现快照，这似乎成了一个不成文的行业标准，不能实现快照的产品无法在市场上生存。目前市场上也有很多基于 NTFS 文件系统的第三方独立快照管理软件，例如《还原精灵》系列，以及《雨过天晴》系列等，它们都是非常优秀的基于 NTFS 文件系统的快照管理软件。这些软件的作用方式一般是将自己的程序入口写入 MBR，系统引导初期会首先加载这些程序，关机之后始终在操作系统下层运行，这样就可以肆无忌惮的接管操作系统对 NTFS 文件系统的所有操作。在这个基础上，程序可以实现文件系统的快照以及管理这些快照。目前《雨过天晴》提供 7 天的试用版，感兴趣的朋友可以下载试用。

16.2.4 Continuous Data Protect(CDP，连续数据保护)

SNIA 对于 CDP 给出了一个定义。CDP，持续数据保护是一种在不影响主要数据运行的前提下，可以实现持续捕捉或跟踪目标数据所发生的任何改变，并且能够恢复到此前任意时间点的方法。CDP 系统能够提供块级、文件级和应用级的备份。

有一类所谓的 Near CDP 产品，可以生成高频率的快照，比如一小时几十次，上百次等。用这种方法来保证数据恢复的粒度足够细。

CDP 是这样一种机制，即它可以保护从某时刻开始，卷或者文件在此后任意时刻的数据状态，也就是数据的每次改变，都会被记录下来，无一遗漏。这个机制乍一看非常神奇，其实它的底层只不过是比快照多了一些考虑而已，下面我们就来分析它的实现原理。

文件级的 CDP

顾名思义，文件级 CDP 就是通过调用文件系统的相关函数，监视文件系统动作，文件的每一次变化，都会被记录下来。这个功能是分析应用对文件系统的 IO 数据流，然后计算出文件变化的部分，将其保存在 CDP 仓库设备(存放 CDP 数据的介质)中。每次对文件的改变，都会被记录下来。可以对一个文件，或者一个目录，甚至一个卷来监控。文件级的 CDP 方案，一般需要在生产主机上安装代理，用来监控文件系统 IO，并将变化的数据信息传送到 CDP 仓库介质中。文件级的 CDP，能够保证数据的一致性。因为它是作用于文件系统层次，捕获的是完整事物。

块级的 CDP

块级的 CDP，就是捕获底层卷的写 IO 变化，并将每次变化的块数据保存下来。这里不探讨具体产品的架构，而只对其底层原理，作一个细致的描述。



以下对于 CDP 的描述引自敖青云的博客(aoqingy.spaces.live.com)。

CDP 起源于 Linux 下的 CDP 模块。它持续地捕获所有 I/O 请求，并且在这些请求打上时间戳标志。它将数据变化以及时间戳保存下来，以便恢复到过去的任意时刻。

在 Linux 的 CDP 实现中，包含下列三个设备。

- 主机磁盘设备(host disk)。
- CDP 仓库设备(repository)。
- CDP 元数据设备(metadata)。

CDP 代码对主机磁盘设备在任意时刻所作的写操作都会记录下来，将实体数据按顺序写入 CDP 仓库设备中，对于这些实体数据块的描述信息，则被写入到 CDP 元数据设备的对应扇区。

元数据包含以下信息：

```
struct metadata {
    int hrs, min, sec; 该数据块被写入主机磁盘设备的时间
    unsigned int bisize; 该数据块以字节为单位的长度
```

```
    sector_t CDP_sector; CDP 仓库设备中对应数据块的起始扇区编号
    sector_t host_sector; 该数据块在主机磁盘设备中的起始扇区编号
};
```

图 16.2 反映了主机磁盘设备和 CDP 仓库设备之间的关系。CDP 仓库设备中按时间顺序保存了对主机磁盘设备的数据修改。A 为主机磁盘设备上的一个扇区，该扇区在 9:00 和 9:05 分别进行了修改，它在 CDP 仓库设备中对应的扇区分别为 A1 和 A2。

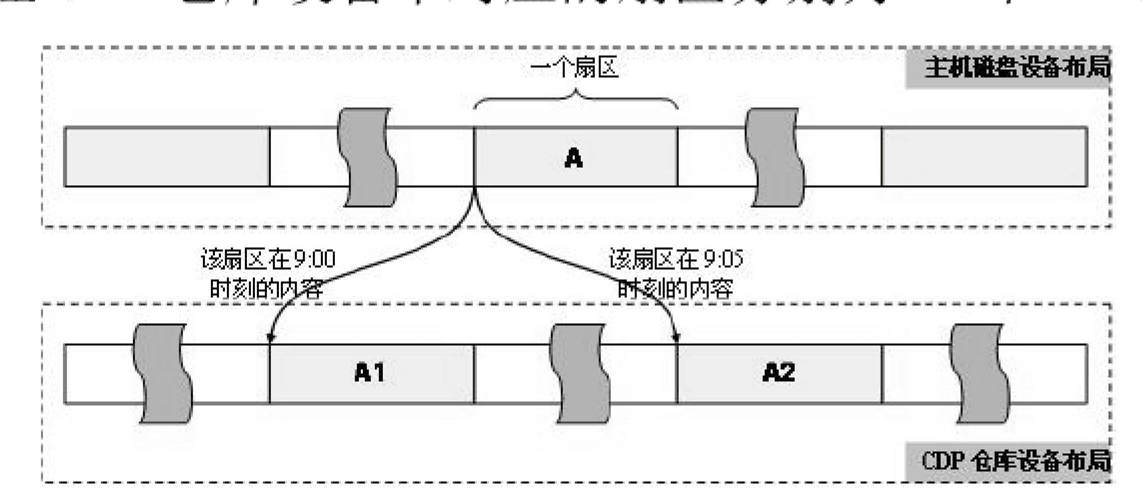


图16.2　CDP 仓库与主机磁盘设备间的关系

图 16.3 反映了 CDP 仓库设备和 CDP 元数据设备之间的关系。它们的写入顺序一一对应。CDP 仓库设备中的一个元数据，对应 CDP 元数据设备中一个 I/O 请求，实际上可能是多个扇区。具体扇区数由元数据中的 bisize 指定，而起始扇区位置由 CDP_sector 指定。

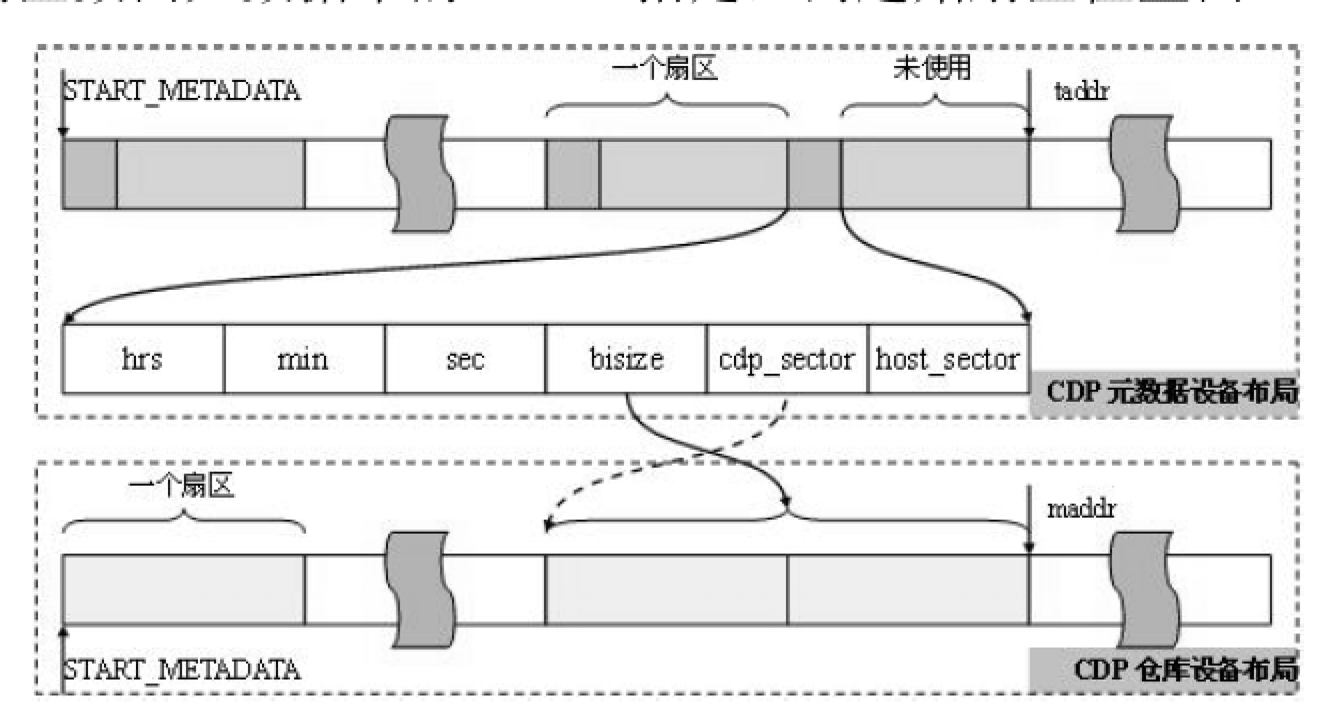


图16.3　CDP 仓库设备与 CDP 元数据设备间的关系

全局变量 maddr 保存了下一个 I/O 请求，在 CDP 仓库设备上执行的地址(起始扇区编号)。maddr 的初值被定义为宏 START_METADATA(0)。

```
unsigned int maddr = START_METADATA;
```

当一个写请求到来时，对应数据被写到 CDP 仓库设备中，这时所做的操作如下。

1】 将写入 CDP 仓库设备的数据块起始扇区编号设置为 maddr。

2】 根据要写入主机磁盘设备的数据块的扇区数目增加 maddr。

这时，用户要将这里写入 CDP 仓库设备的数据块编号记录下来，以便构造对应的元数据。

全局变量 taddr 保存了下一个 I/O 请求，对应的元数据，在 CDP 元数据设备中保存的地址(起始扇区编号)。 taddr 的初值被定义为宏 START_METADATA(0)。

```
unsigned int taddr = START_METADATA;
```

当一个写请求到来时，对应的元数据被记录在 CDP 元数据设备中。

为了简单起见，在元数据设备上，一个扇区(512 字节)只保存一个元数据信息(只有 32

字节)，这样浪费了大量的存储空间，但对元数据设备的处理却非常简单。

1】 将写入 CDP 元数据设备的元数据起始扇区编号设置为 taddr，长度为 1 个扇区。

2】 将 taddr 增加 1。

3】 请求处理过程。

请求处理过程是从 make_request 函数开始的。考虑到读请求处理的相似性，甚至更为简单。这里只分析对写请求的处理过程。首先获得当前的系统时间，然后写请求 bio 结构(为便于说明这里记为 B)被分为三个写请求 bio 结构(分别为 B0、B1 和 B2)，如图 16.4 所示。这三个 bio 结构的作用是。

- B0：将数据块写到主机磁盘设备。
- B1：将数据块写到 CDP 仓库设备。
- B2：将元数据写到 CDP 元数据设备。

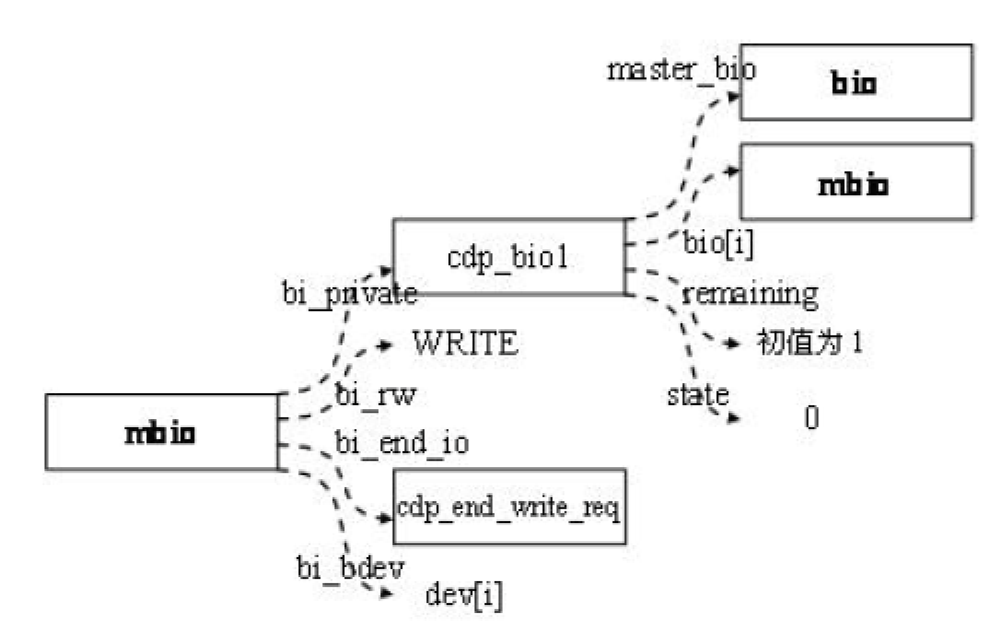


图 16.4 三个写请求 bio 结构

同其他块设备驱动程序的实现一样。从 B 复制产生 B0、B1 和 B2。然后重定向它们要处理的设备，即 bi_bdev 域。另外一个大的变动是重新设置了 bi_end_io 域，用于在 I/O 请求完成之后进行善后处理。

为了处理善后，还要将 B0、B1 和 B2 的 bi_private 指向同一个 CDP_bio1 结构。从这个结构能够回到对 B 的处理。

```
struct CDP_bio {
struct bio *master_bio; 原来的 bio，通过这个域用户可以从 B0、B1、B2 找到 B
struct bio *bios[3]; 如果 IO 为 WRITE，这个指针数组分别指向 B0、B1、B2，为何需要这个域？
atomic_t remaining; 这是一个计数器，在后面会解释。
unsigned long state; 在 I/O 完成方法中使用
};
```

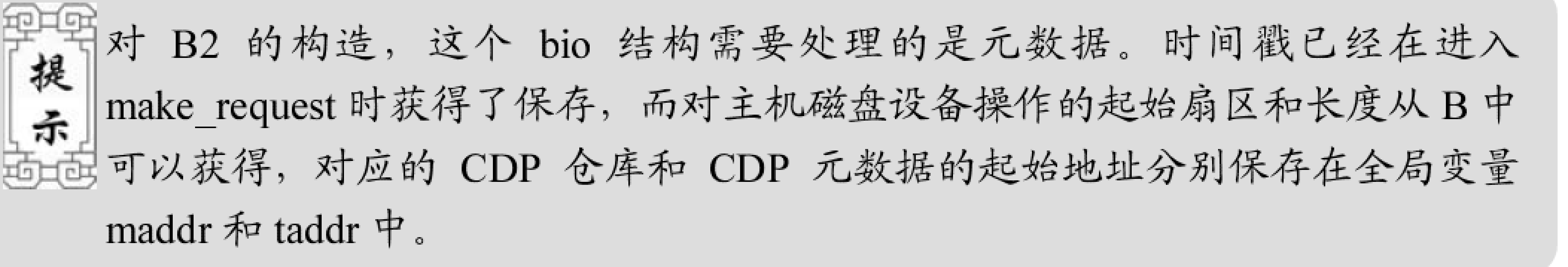
善后工作的主要目的是在 B0、B1 和 B2 都执行完成后，回去执行 B，为此需要一个“have we finished”计数器，这就是原子整型变量 remaining。在构造 B0、B1、B2 时分别递增，同时在 B0、B1 和 B2 的 I/O 完成方法中递减，最后根据该值是否递减到 0，来判断 B0、B1 和 B2 是否都已经执行完毕。为了防止 B0 在构造后，在 B1 和 B2 构造之前就执行到 B0 的 I/O 完成方法，从而使得 remaining 变成 0 这种错误情况。我们没有将 remaining 的初值设置为 0，而是设为 1。并在 B0、B1、B2 都构造完成执行递减一次。

B0、B1、B2 都执行完成之后，进行如下的处理。

1】 调用 B 的善后处理函数。

2】 释放期间分配的数据结构。

3】 向上层 buffer cache 返回成功/错误码。


提示 对 B2 的构造，这个 bio 结构需要处理的是元数据。时间戳已经在进入 make_request 时获得了保存，而对主机磁盘设备操作的起始扇区和长度从 B 中可以获得，对应的 CDP 仓库和 CDP 元数据的起始地址分别保存在全局变量 maddr 和 taddr 中。

数据恢复过程

用户可以将数据恢复到以前的任意时刻。CDP 实现代码中提供了一个 blk_ioctl 函数，用户空间以 GET_TIME 为参数调用该函数，将主机磁盘设备中的数据恢复到指定的时间点。恢复的过程分为以下几个步骤。

1】 顺序读取 CDP 元数据设备的所有扇区，构造一个从主机磁盘设备数据块，到 CDP 仓库设备的(在这个时间点之前)更新数据块的映射。其结果保存在以 mt_home 为首的(映射表)链表中。如图 16.5 所示。

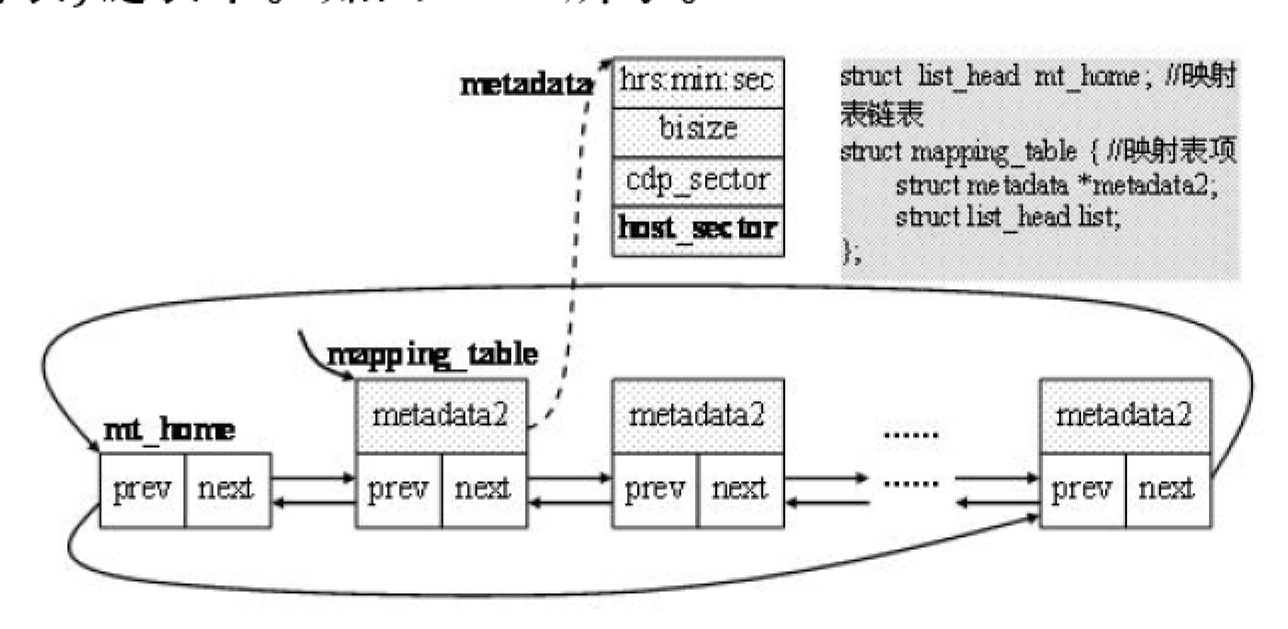


图16.5 CDP 恢复过程

这里需要构造 taddr 个对 CDP 元数据设备的读请求，每个请求读取一个扇区。在这些请求的 I/O 完成方法中，从读到的数据中构造元数据，并递减计数器 count。如果元数据中的时间戳早于或等于指定的恢复时间点，则需要添加或修改 mt_home 链表的元数据结构。

提示 这些项是以 host_sector 为关键字索引的，因此添加或修改取决于前面是否出现对同一个 host_sector 的修改。在以顺序方式读取的过程中，可以保证 host_sector(在指定的恢复时间点之前)的最新修改 CDP_sector 会出现在这个链表中。

由于计数器 count 为 taddr，如果它递减为 0，说明 CDP 元数据设备中的所有数据均已读出并处理，这时就可以继续往后面执行。

2】 从 CDP 仓库设备中读取这些更新的数据块，构造以 mt_bi_home 为首的链表。

同上面的处理类似，我们需要为 mt_home 链表中的每一项构造对 CDP 仓库设备的读请求，每个请求在 CDP 仓库设备的起始编号取决于 CDP_sector 域，长度则根据 bisize 而定。这个请求读出的数据需要被写入到主机磁盘设备中，为此在读请求 I/O 完成函数中，构造一个对应的往主机磁盘设备的写请求 bio，该写请求的起始编号

取决于 host_sector 域，长度根据 bisize 而定，而要写入的数据是刚刚从 CDP 仓库设备中读出的数据。另外，在读请求 I/O 完成函数中，还要递减一个计数器，当该计数器递减到 0 时，说明用户已经全部处理了 mt_home 链表中的项，这时会得到一个以 mr_bio_home 为首，每项中都指向一个 bio 结构的链表。

```
struct list_head mt_home; //BIO 更新链表
struct most_recent_blocks { //BIO 更新表项
    struct bio *mrbio;
    struct list_head list;
};
```

3】 将 mt_bi_home 链表的数据块都恢复到主机磁盘设备中。

这个操作相对比较简单，用户只需要在主机磁盘设备上，执行 mt_bi_home 链表的每一个 bio 请求项即可。当然还要在这些请求项的 I/O 完成方法中做善后处理，即如果所有请求项都已经执行完毕，则释放 mt_home 链表和 mt_bi_home 链表。

16.3 数据备份系统的基本要件

- 备份主体：是指需要对其进行备份的备份源，比如一台服务器上某块磁盘上的所有数据，或者某数据库下所有数据文件，这些都算备份主体。
- 备份目的：是指将备份主体的数据备份到何处。备份目的可以是备份主体本身的磁盘、磁带等介质，也可以是任何其他地点的磁盘、磁带机、磁带库等介质。如果备份目的位于备份主体本身，比如，从一台 Windows 服务器的 D 盘复制某些文件到自身的 E 盘，则不需要占用任何网络资源，因为数据从备份主体自身生成，到自身结束。如果备份目的位于其他地点，比如同一个机房内的其他服务器，或者外部独立磁盘阵列，则数据从备份主体生成，传输到备份目的的过程中，就需要占用网络资源，因为连接备份主体和备份目的的只有网络，稍后将会详细介绍用哪种网络。
- 备份通路：也就是我们上文提到的，如果备份主体和备份目的都位于同一个角色上，那么备份通路就是这个角色自身的计算机总线，也就是连接这个角色的 CPU、内存和磁盘的总线，因为数据只在这个总线上流动。如果备份主体和备份目的处于远离状态，则二者必须通过某种网络连接起来，而这个网络就是这个备份环境的备份通路。关于备份通路会在后面的例子中详细描述。
- 备份执行引擎：有了备份源、备份目的和连接二者的备份通路之后，需要一个引擎来推动数据从备份源流到备份目的。这个引擎，一般由备份软件来担任。
- 备份策略：是指备份引擎的工作规则。引擎不能无时无刻的运转，它需要根据设定好的规则来运转。

下面将着重介绍一下备份目的、备份通路和备份引擎这三个要件。

16.3.1 备份目的

1. 用本地磁盘作为备份目的

用本地磁盘作为备份目的，就是把本地磁盘上待备份的数据，备份到本地磁盘其他的分区或者目录。用这种方式可以不影响任何其他服务器以及共用网络。数据流动的范围完全局限在备份主体自身。

但是这样做的缺点就是对备份主体自身的性能影响太大，数据从磁盘读出，需要耗费磁盘资源。读出后写入内存，需要耗费内存资源，然后再从内存写入磁盘的其他分区，同样需要耗费磁盘资源。备份执行期间，还会对其他 IO 密集型的程序造成极大影响。通常这种方式只用于不太关键的应用和非 IO 密集型应用，以及对实时性要求不高的应用。E-mail 服务器的备份就是典型的例子，因为 E-mail 转发实时性要求不高，转发速度慢一些，对用户造成的影响也不会很大。

2. 用 SAN 磁盘作为备份目的

用 SAN 上的磁盘做备份，就是把备份主体上需要备份的数据，从本地磁盘读入内存，然后从内存中写入连接到 SAN 的适配器，即 HBA 卡缓冲区，HBA 卡再通过线缆，将数据通过 SAN 网络传送到磁盘阵列上。

这种方式的优点就是数据从本地磁盘读出数据，写入的时候只耗费 SAN 共用网络带宽资源，而且能获得 SAN 的高速度，对备份主体性能影响相对较小。缺点是对公共网络资源和盘阵出口带宽有一定影响，因为耗费了一定的带宽用来传输数据，同时数据在流向盘阵接口的时候，也要占用接口带宽。

如果备份主体数据本身就存放在 SAN 上的磁盘，而备份目的同样是 SAN 磁盘，那么数据流动的通路是比较长。在后面介绍备份通路的时候再做详细阐释。

3. 用 NAS 目录作为备份目的

用 NAS 作为备份目的，就是将本地磁盘上的数据备份到一个远程计算机的共享目录中。比如 Windows 环境下常用的文件夹共享，就是这样一个典型的例子。一台计算机共享一个目录，另一台计算机向这个目录中写入数据。而数据一般是通过以太网络来进行传递的。这种方式占用了前端网络的带宽，但是相对廉价，因为不需要部署 SAN。

4. 用 SAN 上的磁带库作为备份目标

用 FC 接口作为外部传输接口的设备，不仅仅有主机上的 HBA 适配器、磁盘阵列，磁带机和磁带库也可以用 FC 接口作为外部传输接口。用线缆连接磁带库和 SAN 交换机之后，处于 SAN 上的所有主机系统便会识别出这台磁带库设备，自然也就可以用磁带来当作备份目的了。

磁带库，由机械手、驱动器、磁带槽组成。图 16.6 和图 16.7 分别为某型号磁带库的外视图和某型号磁带库的内视图。



图16.6　某型号磁带库的外视图



图16.7　某型号磁带库的内视图

磁带驱动器是磁带库的核心组件。可以将驱动器想象成一个电机，带动磁带旋转，然后磁头贴住磁带、读写数据，把电机、磁头以及控制电路集合到一起，形成一个独立的模块，就是驱动器。

机械手，在机械生产线上机械手就是一个计算机夹子而已，把物品从一个地方移动到另一个地方，当然这还需要程序来控制。而且像图 16.7 中所示的一样，机械手不一定就是那种铁臂抓手，而只要它能寻找磁带槽上的磁带并将其推入磁带驱动器，就可以称其为机械手。

磁带库的工作流程如下：

1】 **由机械手臂从磁带槽中夹取一盘磁带，推入磁带驱动器，驱动器完成倒带、读写等动作。**

2】 **读出完成后，退带，机械手臂夹取磁带，放回磁带槽，然后夹取另一盘磁带放入驱动器，重复刚才的动作。**

整个流程都需要由程序来控制机械手臂和驱动器，那么程序运行在哪里呢？当然是连接磁带库的主机上，程序生成符合协议的电信号，经过 HBA 卡传送到磁带库电路板上，经过芯片处理，转换成操控机械手臂和驱动器的另一串电信号。所以连接磁带库的主机上，除了需要安装 HBA 适配器的驱动程序之外，还需要安装磁带库机械手和驱动器的驱动程序，这样才能够按照驱动程序定义的规则，来生成符合规定的、磁带库可以识别的电信号。

磁带机比磁带库功能少，但是基本原理都是一样的，只不过机械手臂没了，取磁带和放磁带需要用人手而已。而且一台磁带机同一时刻只能操作一盘磁带。而在磁带库中可能有多个驱动器，或者多个机械手，当然机械手不需要那么多，因为一个机械手就能完成，除非驱动器多得让一个机械手都忙不过来，这是不太可能发生的事情了。多个驱动器可以同时读写多盘磁带(每个驱动器一盘)，使得效率大大提升。



近年来，出现了一种虚拟磁带库产品，即用磁盘来模拟磁带。当然，磁盘就是磁盘，不可能变为磁带，那磁盘是怎么被虚拟成磁带的呢？当然是通过存储控制器来虚拟化。

话说回来是磁带还是磁盘？这都取决于数据服务器看到的影像，而它看到的也不一定就是实际上存在的。虚拟磁带库也正是利用了这个原理。使用磁带库的是主机服务器，如

果让主机服务器看到的影像就是一个磁带库，而实际上却是一台磁盘阵列，那么主机照样会像使用磁带库一样使用这台虚拟的磁带库。要做到这一点，就必须在磁盘阵列的控制器上做虚拟化操作，也就是要实现协议转换器类似的作用，一边以磁带库的逻辑工作，另一边以磁盘阵列的逻辑工作。

虚拟磁带库的好处。

- 速度大大提升。因为向磁盘写入数据要比磁带快。
- 避免了机械手这种复杂的机械装置，取而代之的是控制器上的电路板。
- 管理方便，随意增删虚拟磁带。



有了一个虚拟磁带库的壳子还不够，很多厂商还在虚拟磁带库上开发了增值软件，这些软件形成了一个很大的存储课题，就是所谓的分级存储或者存储生命周期管理。

5. 信息生命周期管理

假如，某企业有磁盘阵列容量共 1TB，磁带库容量 10TB。每天均会新生成 5GB 左右的视频文件，这些视频文件都存放在磁盘阵列的一个 500GB 的卷中。这样不到 3 个月，这 500GB 的卷便会被全部用完。而这个企业频繁调出查看的视频，一般都是最近一个月以内的，如果将一个月之前的、几乎很少或者永远也不会被再次访问到的视频也放在磁盘阵列上，这无疑是一个巨大的浪费，因此完全可以把这些文件备份到磁带库，而腾出磁盘阵列上的存储空间，供其他应用程序存储数据。

磁盘阵列是高速数据存储设备，而磁带库是低速数据存储设备，所以为了各得其所、物尽其用，有人便开发了一套信息生命周期管理软件，这种软件根据用户设定的策略，将使用不频繁的数据，移动到低速、低成本的存储设备上。比如只给某个视频应用分配 20GB 的磁盘阵列的空间，但是向它报告 500GB 的存储空间，其中有 480GB 其实是在磁带库上的。

这样，应用程序源源不断的生成视频数据，而管理软件根绝策略，比如某视频文件超过了设定的存留期，便将它移动到磁带库上，腾出磁盘阵列上的空间。但是对于应用程序来说，总的可用空间还是在不断的减少。虽然磁盘阵列上可能总是有空间，但这些空间是给最近生成的文件使用的，因为这些文件会被频繁访问。如果一旦需要访问已经被移动到磁带库上的文件，则管理软件会从磁带库提取文件，并复制到磁盘阵列上，然后供应用程序访问。

6. 分级存储

基于信息生命周期管理的这个思想，目前很多厂家都在做相应的解决方案，分级存储就是这样一种方案。

- 第一级：一线磁盘阵列，是指存储应用频繁访问数据的磁盘阵列。其性能相对二线和三线设备来说应该是最高的。
- 第二级：二线虚拟磁带库。这个级别上的存储设备，专门存放那些近期不会被频繁访问的数据。其性能和成本应该比一线设备低，但是性能不至于太低，以致于提取数据的时候造成应用长时间等待。所以虚拟磁带库，正好满足了这个要求。虚拟磁带库利用成本比较低廉的大容量 SATA 磁盘，性能适中的存储控制器，这样

保证了性能不至于像磁带库一样低，成本又不会像一线设备一样高。

- 第三级：磁带库或者光盘库等。这个级别上的设备，专门存储那些几年甚至十几年都不被访问到的，但是必须保留的数据。磁带库正好满足了这个要求，这是毫无疑问的。

16.3.2 备份通路

1. 本地备份

本地备份的数据流向是：本地磁盘－总线－磁盘控制器－总线－内存－总线－磁盘控制器－总线－本地磁盘。即数据从本地磁盘出发，通过本地的总线和内存，经过 CPU 运算少量控制逻辑代码之后，最终流回本地磁盘。

2. 通过前端网络备份

通过前端网络备份的数据流向是：本地磁盘－总线－磁盘控制器－总线－内存－总线－以太网卡－网线－以太网络－网线－目标计算机的网卡－总线－内存－总线－目标计算机的磁盘。即数据从本地磁盘发出，流经本地总线和内存，然后流到本地网卡，通过网络传送到目标计算机的磁盘上。

这里说的前端网络，指的是服务器接受客户端连接的网络，也就是所谓“服务网络”，因为这个网络是服务器和客户端连接的必经之路。

后端网络，是对客户封闭的，客户的连接不用经过这个网络，后端网络专用于服务器及其必须的后端部件之间的连接，比如，和存储设备，或者应用服务器和数据库服务器之间的连接，这些都不需要让客户终端知道。

后端网络可以是 SAN，也可以是以太网，或者其他任何网络形式。以太网并不一定就特指前端网络，也可以用于后端。随着以太网速度的不断提高，现在已经达到了 10Gb/s 的速率，所以以太网 LAN 作为后端网络，同样也是有竞争力的。但是说到 SAN，一般就是特指后端网络。

3. 通过后端网络备份

通过后端网络备份的数据流向是：本地磁盘－总线－磁盘控制器－总线－内存－总线－后端网络适配器－线缆－后端网络交换设施－线缆－备份目的的后端网络适配器－总线－内存－备份目的的磁盘或者磁带。



这里说的“后端网络适配器”，泛指任何形式的后端网络适配器，比如 FC 适配器、以太网卡等。

4. LAN Free 备份

LAN Free 这个词已经在存储领域流行使用多年。它的意思是备份的时候，数据不流经 LAN，也就是不流经前端网络。由于历史原因，导致了人们的思维定势，认为 LAN 只用于前端网络，所以说到了 LAN 就想到了前端网络，然而，我们上文已经做了解释，后端网络同样可以使用以太网 LAN。LAN 这个词本意为“Local area network”，即局域网络，它没

有对网络的类型加以限制，可以说存储区域网络也是一个 LAN。


笔者认为这个词不再适合当今存储领域，取而代之的应该是 Frontend Free 这个新的名词，即备份的时候，数据不需要流经前端网络，而只流经后端网络。

Frontend Free 备份的好处是：不耗费前端网络的带宽，对客户终端接收服务器数据不会造成影响。相对于后端网络来说，前端网络一般为慢速网络，资源非常珍贵，加上前端网络是客户端和服务器端通信的必经之路，所以要尽量避免占用前端网络的资源，备份数据长时间频繁的流过前端网络，无疑会对生产造成影响。解决的办法就是通过后端网络进行备份，或者本地备份。

无论是本地备份，还是通过网络备份(前端网络或者后端网络)，都需要待备份的服务器付出代价来执行备份，即服务器需要读取备份源数据到自身的内存，然后再从内存将数据写入备份目的，对主机 CPU、内存都有资源耗费。是否能让服务器付出极小的代价，甚至无代价而完成备份任务呢？当然可以。

5. Server free 备份

这个名词是指，备份的时候，数据甚至不用流经服务器的总线和内存，消耗极少，甚至不消耗主机资源。下面来分析一下。

要想使备份数据不流经服务器本身，那么首先备份主体，即待备份数据所在的地方肯定不能是服务器的本地磁盘，因为数据从磁盘读出，第一个要流经的地方就是总线，然后到服务器内存，这样就不叫 Server free 了。

所以，备份源不能在服务器上，同理，备份目的也不能在服务器上，不然写入的时候照样流经服务器的总线和内存。那么到底怎么样才能实现 Server free 呢？

很简单，备份主体和备份目的都不在服务器上，不在本地就只能在 SAN 上了。做到这一点还不够，因为主机要从 SAN 上的一个磁盘取出数据，写入 SAN 上的另一个磁盘，同样需要先将数据读入到主机的内存，然后再写入 SAN。那么到底怎么样才能做到 Server free 呢？

答案是，用 SCSI 的扩展复制命令，将这些命令发送给支持 Server free 的存储设备，然后这些设备就会提取自身的数据直接写入备份目的设备而不是发送给主机。或者用另一台计算机作为专门移动数据之用，即待备份的主机向这台数据移动器发信号，告诉它移动某磁盘上的数据到另一个磁盘，然后这台数据移动器从 SAN 上的源磁盘读取数据到它自己的内存而不是待备份主机的内存，然后写入到 SAN 上的目标磁盘。


所谓的 server free，并不是真正的不需要服务器来移动数据，而是让服务器发出扩展复制命令，或者使用另一台专门用作数据移动的新服务器，来代替原来服务器移动备份数据，释放运算压力很大的生产服务器，当然，SAN 上的源磁盘和目标磁盘或者磁带，数据移动服务器都需要有访问权。

为了统一数据备份系统中所有节点之间的消息流格式，Netapp 公司和 Legato 公司合作

开发了一种叫做 NDMP 的协议(网络数据管理协议)。这个协议用于规范备份服务器、备份主体、备份目的等备份系统各种节点的数据交互控制。 服务器只要向支持 NDMP 协议的存储设备发送 NDMP 指令，即可让存储设备将其自己的数据直接备份到其他设备上，而根本不需要流经服务器主机。

16.3.3 备份引擎

备份引擎，就是一套策略、一套规则，它决定整个数据备份系统应该怎么运作，按照什么策略来备份，备份哪些内容，什么时候开始备份，备份时间有没有限制，磁带库中的磁带什么时候过期并可以重新抹掉使用等。就像引擎一样，开动之后，整个备份就按照程序有条不紊的进行。

1. 备份服务器

那么备份引擎以一种什么形式来体现呢？毫无疑问，当然是运行在主机上的程序来执行，所以需要有这么一台计算机来做这个引擎的执行者，这台计算机就叫做“备份服务器”，意思就是这台计算机专门管理整个数据备份系统的正常运作，制定各种备份策略。



备份服务器的备份策略和规则，是怎么传送给整个数据备份系统中的各个待备份的服务器呢？和汽车一样，车轮和引擎之间有传动轴连接，备份服务器和待备份的服务器之间也有网络来连接，那么通过以太网还是通过 SAN 网络来连接呢？

答案是以太网络，因为以太网络使用广泛，以太网之上的 TCP/IP 编程已经非常成熟，非常适合节点间通信。相对于以太网，SAN 更加适合传送大量数据。而利用前端网络连接还是利用后端网络连接呢？

一般我们常用前端网络来连接待备份服务器和备份服务器。因为备份策略就好比两个人之间说了几句话，所以把这几句话传送给待备份服务器，不会耗费很大的网络资源，充其量每秒几十个包而已，这对前端网络影响非常小。有了网络连接，我们就有了物理层的保障。

但备份服务器是如何与每个待备份的服务器建立通话的呢？它们之间怎么通话？通话的规则怎么定呢？这就需要待备份服务器上需要运行一个程序，专门解释备份服务器发来的命令，然后根据命令，做出动作。

这个运行在各个待备份服务器上的程序，就叫做**备份代理**(backup agent)，他们监听某个 Socket 端口，接收备份服务器发来的命令.比如，某时刻备份服务器通过以太网前端网络，给某个待备份服务器发送一条命令，这条命令被运行在该待备份服务器上的备份代理程序接收，内容是：立即将位于该服务器上的 C 盘下 XX 目录拷贝到 E 盘下 XXX 目录。备份代理接收到这个命令之后，就会将该待备份服务器上 C 盘下 XX 目录拷贝到 E 盘下的 XXX 目录。如果 D 盘是本地盘，E 盘是一个 SAN 上的虚拟磁盘，那么实际数据流动的路径，就是：本地磁盘－总线－内存－总线－SAN 网络适配器－线缆－SAN 交换设施(如果有)－磁盘阵列。

数据源源不断地从本地磁盘流向 SAN 网络上的磁盘阵列，成功备份之后，备份代理收

到来自待备份服务器操作系统的成功提示，然后备份代理通过以太网向备份服务器返回一条成功完成的提示。这样备份服务器便会知道这个备份已经成功完成，并记录下开始时间、结束时间、是否成功等信息。

同样的道理，如果例子中的 E 盘也是本地盘，那么路径就短多了。如果备份服务器告诉备份代理，将数据拷贝到位于 SAN 上的磁带库设备而不是磁盘阵列，那么同样，备份代理将数据从本地磁盘读出，然后通过 SAN 网络适配器发往磁带库。当然这需要在待备份的服务器上安装可以操控磁带库设备的驱动程序。同理 E 盘如果是个 NAS 目录，则数据便会被发往远端的 NAS 服务器了。

2. 介质服务器



假如在一个数据备份系统中，有一台普通的 SCSI 磁带机连接在某台主机上，且有多台主机的数据需要备份到这台 SCSI 磁带机的磁带中，而 SCSI 磁带机只能同时接到一台主机上，总不能搬着磁带机，给每个机器轮流插上用吧？

当然可以，但是这样很麻烦。有没有一种办法来解决这个问题呢？当然有了。可以将这台磁带机连接到固定的一台计算机，只能由这台计算机来操作磁带机。其他有数据需要备份的计算机，和这台掌管 SCSI 磁带机使用权的计算机，通过以太网连接起来(当然也可以通过其他网络方式连接，但前面说过，以太网是最廉价最广泛使用的网络)，谁有数据，谁就将数据通过以太网发给这台掌管磁带机的计算机，收到数据后，这台计算机将数据写入只有它才有权控制的磁带机。写完后，下一台有数据需要备份的计算机，重复刚才的动作。

这样，我们用了一台计算机来掌管 SCSI 磁带机，然后在这台计算机的前端，我们用以太网扩展了连接。虽然微观上磁带机只有一个接口，只能连接一台计算机，但是经过以太网的扩展之后，这台磁带机成为了公用设备，掌管磁带机的计算机成为了代替这些服务器行使备份动作的角色，因为整个数据备份系统中，只有这台计算机掌管了备份目的，也就是磁带机、磁带，所以我们称这台服务器为“介质服务器”，也就是说，这台服务器是数据备份系统中备份介质的掌管者，其他人都不能直接访问备份介质。



这台计算机只是掌握了备份介质，谁都可以向它发起请求，然后传输需要备份的数据给它，但是如果同时有多台服务器向它发出请求，怎么办？

显然还需要一个调度员，来管理调度好多个待备份服务器之间的顺序，做到有条不紊，按照预订的策略来备份，避免冲突。那么谁来担当调度员呢？前面讲到的“备份服务器”本身就是这样一个调度。所以非它莫属了。

将这个调度员也接入以太网，调度员使出它的必胜大法——在每个待备份的服务器上，都安装它的“耳朵”和“嘴巴”，即备份代理程序，通过这个耳朵和嘴巴，调度员让每台服务器都乖乖的听话，按照顺序有条理的使用介质服务器提供的备份介质进行备份。在一个数据备份系统中，介质服务器可以有多台同时分担工作。

图 16.8 是一个 Veritas Netbackup 备份软件的备份流程。

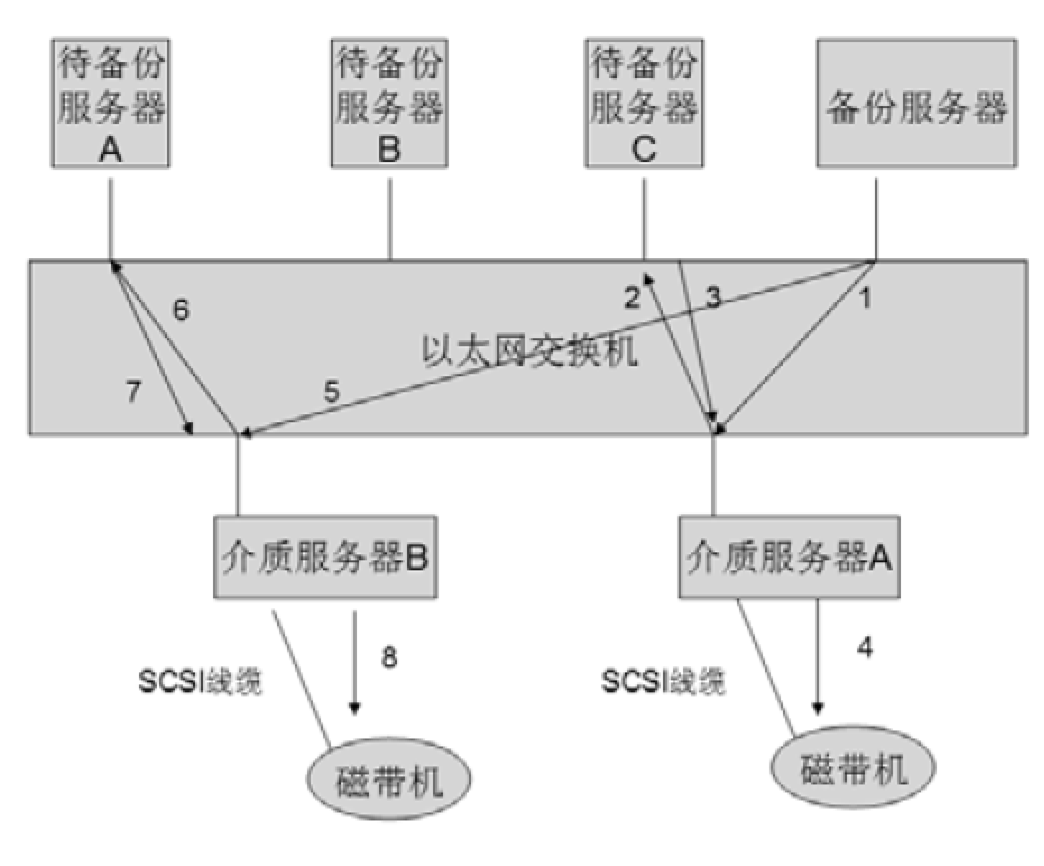


图16.8 备份流程

1】 **某时刻，备份服务器发起备份，它通知“介质服务器 A”备份“待备份服务器 C”上的相应内容。**

2】 **“介质服务器 A”向服务器发出指令，告诉它可以进行备份了，请发送需要备份的数据。**

3】 **待备份服务器 C 把需要备份的数据通过以太网发送给“介质服务器 A”。**

4】 **“介质服务器 A”将收到的数据源源不断的写入磁带机。**

5】 **重复第 1 步，只不过介质服务器为 B，待备份服务器为 A。**

上面这个拓扑图，是一个 Frontend unfree 备份方式。因为备份数据流占用了前端网络带宽。

至此，我们的数据备份系统中，已经有了三个角色了：备份服务器(调度员)、介质服务器(仓库房间管理员)，待备份服务器(存储货物的人)。

再转回去看看还没有出现“介质服务器”前的那个例子。会发现那时候，仓库房间尚充足，仓库每个门都开着，每个人都可以从各自的门进去存放物品，每个需要存放货物的人都有权直接访问它们的仓库房间，它们只靠一个调度员来协调，这时候可以认为，每个消费者都在管理着仓库房间，它们每个人都是仓库房间管理员，只不过它们各自管理自己的房间而已。所以，这种情况下，每台待备份的服务器，都是介质服务器，而且每台介质服务器因为都需要操控备份设备，所以还需要安装诸如磁带库等设备的驱动程序。而备份服务器只有一个。

相对于由存放货物的人自行管理仓库房间的情况，由专人来管理仓库房间，所耗费的前端网络资源更大。因为存储货物的人，首先需要通过网络将货物发送给仓库管理员，然后管理员再将货物放入仓库。而如果让存储货物的人自己存放，则会省去第一步。

图 16.9 所示的就是用上述思想进行数据备份的一个拓扑图。

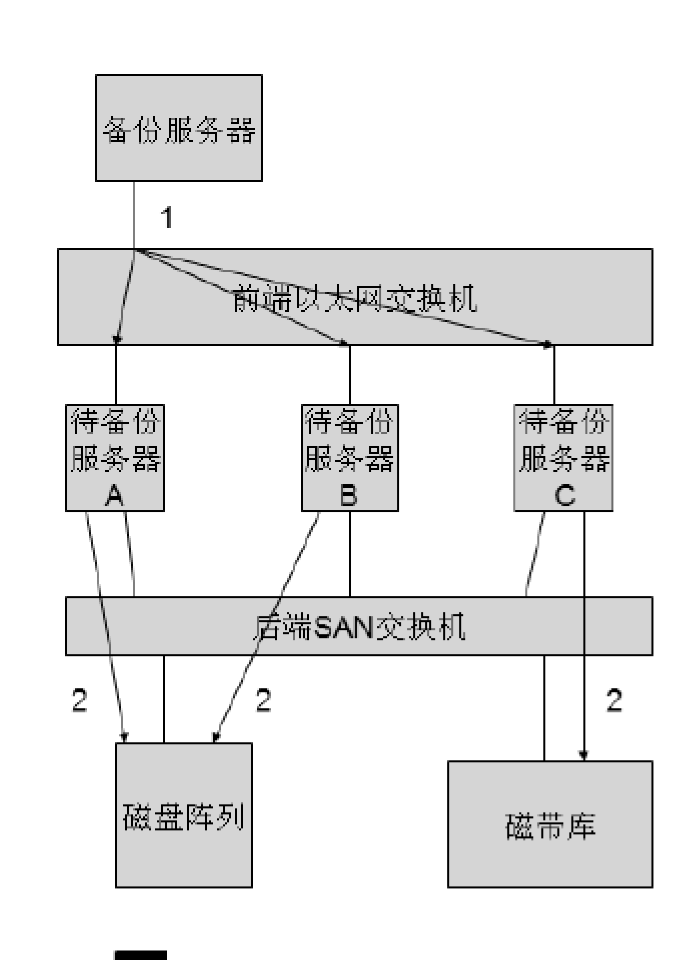


图16.9 Frontend Free

1】 **备份服务器通过以太网，同时向三台介质服务器**

(也是待备份服务器)发出备份开始指令。

2】 待备份服务器直接将数据通过后端 SAN 网络设施写入备份目的。A 和 B 的备份目的是磁盘阵列，它们可以同时写入备份目的。C 的备份目的是磁带库，如果磁带库只有一个驱动器，则同一时刻只能用于一个备份操作，这个例子中，磁带库被 C 独占。当然 C 完成备份之后，磁带库可以被其他服务器使用。A、B、C 都安装有磁带库机械手和驱动器的驱动程序。

上面的拓扑图(图 16.9)，就是一个典型的 Frontend Free 备份方式，因为备份数据流不占用前端网络带宽，而只有备份时候所发送的指令数据经过了前端网络。

随着各种应用系统的不断出现，比如，各种数据库管理系统、E-mail 转发处理系统、ERP 系统、办公自动化系统等，备份技术也随之飞快发展，传统的备份操作，仅仅是备份操作系统文件，即不管这个文件是何种类型，被什么应用程序生成和使用，备份的时候统统当作一个抽象的备份源来看待，只需要把这个文件整体传送到备份目的就可以了。

如今，用户的需求越来越高，越来越细化。比如，某用户要求只备份某数据库中的某个表空间，或者只备份某个 E-mail。由于一个表空间包含一个或者多个数据文件，这样命令发给备份代理的时候，只能是这样："备份某数据库下的某表空间"，而不是："备份 X 盘 X 目录下 XXX 文件"。因为调度员不可能知道某个表空间到底包含哪些具体的文件。要完成这个备份动作，必须由运行在待备份服务器上的某种代理程序来参与，那么这个代理是否可以是上文提到的备份代理呢？

完全可以，但是有个需要增加的功能，即这个代理程序必须可以与待备份的应用程序进行通信，从而获得相关信息。可以有两种方式来完成备份：直接获取到这个表空间对应的数据文件有哪些，然后自行备份这些文件；或者直接调用数据库管理系统的命令，向数据库管理系统发出命令，备份这个表空间。

如果是第一种方式，将调度员发出的命令对应成实际文件的工作，比如，需要备份的是一个 DB2 数据库上的某个表空间，那么这个代理程序需要与 DB2 实例程序进行通信，DB2 实例服务程序告诉备份代理，XXX 表空间对应的容器(数据文件)为某某路径下某某文件。备份代理获得这些信息后，直接将对应的文件备份到备份目的。

如果是第二种方式，备份代理收到调度员指令之后，便向会 DB2 实例服务程序发出指令："db2 backup db testdb tablespace userspace1 online to \\.tape0"，这样，就利用 DB2 自身的备份工具备份了数据。目前广泛使用的做法都是第二种方式，因为解铃还需系铃人，用应用程序本身的备份工具进行备份是最保险的方法。

综上所述，对于每一种待备份的应用程序，都需要一个可以和该应用程序进行通信的代理程序，这就需要开发针对各种应用程序的代理。目前，像 Symantec Backup Exec 备份软件，提供了诸如 Oracle 数据库代理、DB2 数据库代理、Exchange 代理、Lotus Notes 代理、SQL Server 代理、SAP 代理等诸多应用程序的备份代理。

图 16.10 中的 ServerFree Option 就是用来实现 ServerFree 功能的一个模块。Agent for 开头的选件，就是针对各种应用程序所开发的备份代理程序。NDMP 选件，用来实现 ServerFree 所需要的协议栈。

SAN Shared Storage Option，是用来管理那种"每台待备份服务器都是介质服务器"情况下，各个服务器对备份目的设备的共享使用权的一个模块，因为每台服务器都装有目标

设备(磁带库等)的驱动程序，都可以控制磁带库，为了避免意外冲突，SAN Shared Storage Option 作为一个附加的选件来协调各个服务器之间有顺序的使用目标设备而不发生冲突。

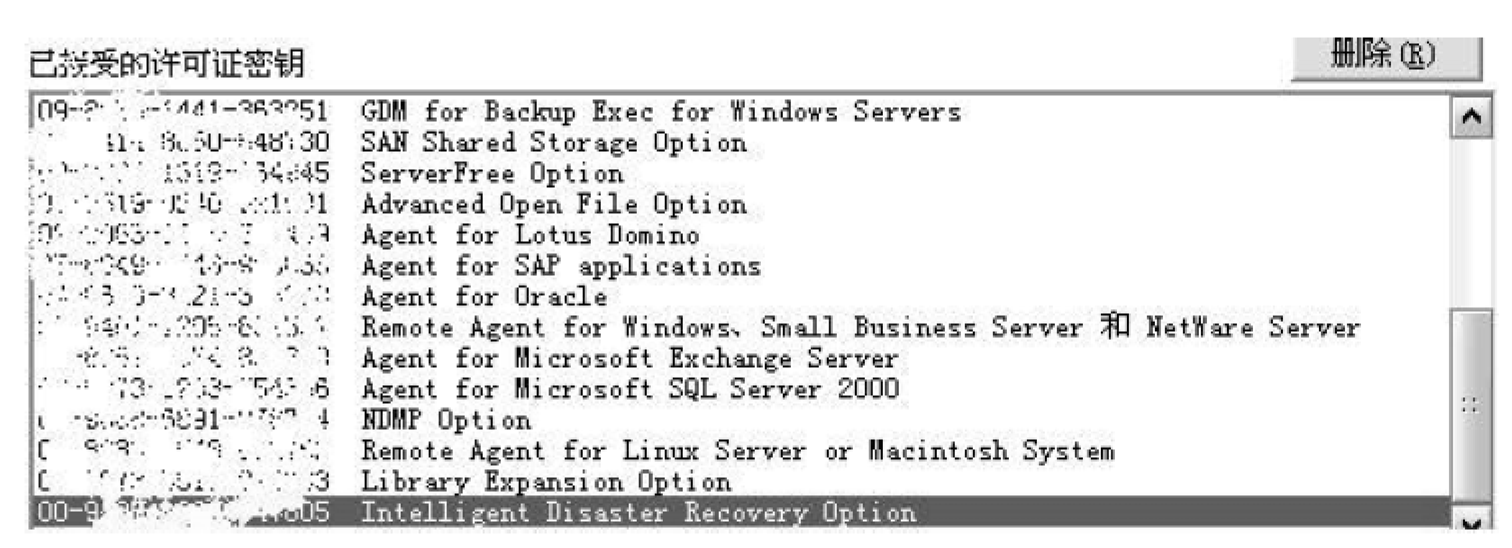


图16.10 各种备份代理及插件

16.3.4 三种备份方式

1. 完全备份

假如某时刻某文件中只包含了一个字符 A。此时我们对这个文件做了备份操作，将其复制到其他介质上，这份备份的文件中只包含字符 A。

稍后，这个文件被修改，在字符 A 之后增加了一个字符 B。此时我们又对这个文件做了备份，将其复制到其他介质上，这份备份的文件中包含字符 A，B。

以后，不管这个文件怎么变化，变成多大，包含多少个字符，只要备份，就将这个文件整个备份下来，这就是完全备份。

2. 差量备份

假如某时刻某文件中包含 100 个数字 1、2、3、…100，在对其做完全备份后，文件中增加了一个数字 101，又过了一段时间，文件中又增加了一个数字 102，而我们已经有了一个完全备份，这个备份已经包含了文件中 1～100 这 100 个数字，如果我们此时再次对这个文件做完全备份，是不是太浪费时间和存储空间了呢？得想出一种办法，即只备份自从上次完全备份以来发生变化的数据。

完全可以，我们以一种自己可以识别的格式，将 101 和 102 这两个数字保存到一个单独的文件中。这就叫做差量备份，意思就是只备份与上次完全备份内容之间相差的内容。如果要恢复这份最新的文件，只要将上次完全备份和最后一次的差量备份合并起来，便可组成最终的最新完全备份，从而恢复数据。

差量备份要求必须对数据做一次完全备份，从而作为差量的基准点。否则随意找一个基准点，所生成的数据是不完整的。

3. 增量备份

经过了完全备份和差量备份，我们已经有了两个备份文件：包含数字 1～100 的完全备份和包含数字 101 和 102 的差量备份。如果这份文件每天都会增加一个数字，而我们每天都要做差量备份甚至完全备份么？这里还有一种选择，就是增量备份。

增量备份是指：只备份自从上次备份以来的这份文件中变化过的数据。这里的“上次备份”，不管上一次备份是全备，差备，还是增备自身，本次增量备份只备份和上一次备

份结束的时刻，这份文件变化过的数据。比如，我们现在拥有包含数字 1～100 的完全备份和包含数字 101 和 102 的差量备份。此时，我们打算以后每天执行增量备份，那么，第二天，这份文件增加了一个数字 103，所以我们只备份 103 这个数字，以次类推，第三天我们只备份 104 这个数字，这样备份速度极大地加快了，备份所消耗的空间也小了。

提示 在实际使用中可以灵活的制定各种策略，比如每周一对数据进行完全备份，周二到周五每天对数据进行增量备份等。

如果对数据进行增量或者差量备份，普通的文件，备份软件一般是可以检测到文件相对上次备份时候所发生的变化的。但数据库的备份，备份软件想检测某个数据文件的变化，一般来说是不可能的，因为这些文件内部格式是非常复杂的，只有数据库管理软件自身才能分析并检测出来，所以每个数据库管理软件(如 Oracle、DB2 等)都有自己的备份工具，可以全备、差备和增备，而第三方备份软件在对数据库做备份的时候，只能调用数据库软件自身提供的各种命令，或者程序接口。

16.3.5 数据备份系统案例一

前面介绍了数据保护的基本原理和大体思路，下面来看下现实中的数据备份领域，了解一下现今广泛实行的数据备份都是怎么做的。

下面用一个企业的 IT 系统作为一个初始化的例子，如图 16.11 所示，某企业 IT 系统现有 FTP 服务器一台，E-mail 服务器一台，基于 SAP 的 ERP 服务器一台，DB2 数据库服务器一台(用于 SAP 服务器的后台数据库)，备份服务器一台，大型 FC 磁盘阵列一台，小型磁带库一台(一个机械手，二个驱动器)。

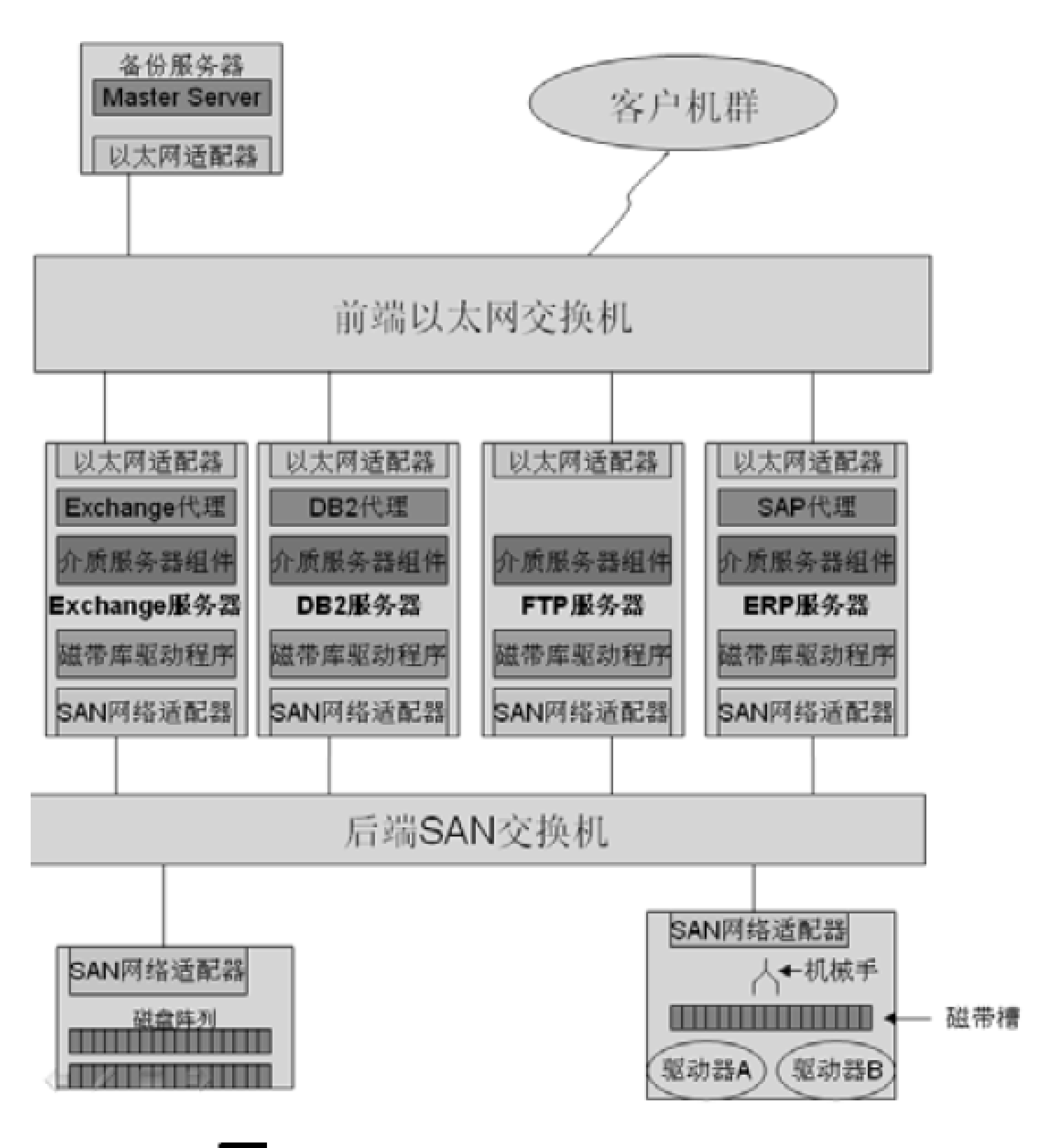


图16.11 某公司 IT 系统示意图

以上五台服务器各用以太网卡连接到同一个以太网交换机上，同时各个办公室的客户端 PC 也通过局域网连接到这台交换机上。另外，除备份服务器之外的四台服务器上分别装有一块 FC HBA 适配卡，且通过光纤连接到一台 SAN 交换机上。磁盘阵列也用一条光纤连接到 SAN 交换机上。该企业使用 Symantec Backup Exec 11d 备份软件进行备份操作。

在备份服务器上需要安装 Symantec Backup Exec 的软件 Master Server 模块。

在每台待备份的服务器上需要安装 Symantec Backup Exec 的 Media Server 模块、磁带库驱动程序、对应的应用程序备份代理模块。

在备份服务器上，用 Symantec Backup Exec 提供的配置界面，来制定针对每台服务器的备份策略，策略生效之后，各个服务器便会按照策略中规定的时间、备份源、备份目的来将各自的数据备份到相应的备份目的。这是一个典型的企业数据备份系统案例，数据流经的路径不包括前端网络，所以属于 Frontend Free 备份。

16.3.6 数据备份系统案例二

Symantec Netbackup 是 Symantec 公司的另一个备份产品，与 Backup Exec 不同的是，Netbackup 适合于大型备份系统，支持各种操作系统平台，各个模块可以分别安装在不同操作系统上，由于之间通过 TCP/IP 协议通信，所以可以屏蔽各种操作系统的不同。而 Backup Exec 只支持 Windows 和 Netware 操作系统(最新的 11D 版本支持 LINUX)。NetBackup 更加适合异构操作系统平台的备份，因此适合拥有众多不同厂家服务器、不同操作系统的大型企业的备份系统。



关于 NetBackup 可参考第 16.3.7 节的 NetBackup 配置指南。

图 16.12 所示的是某企业备份系统的拓扑图。

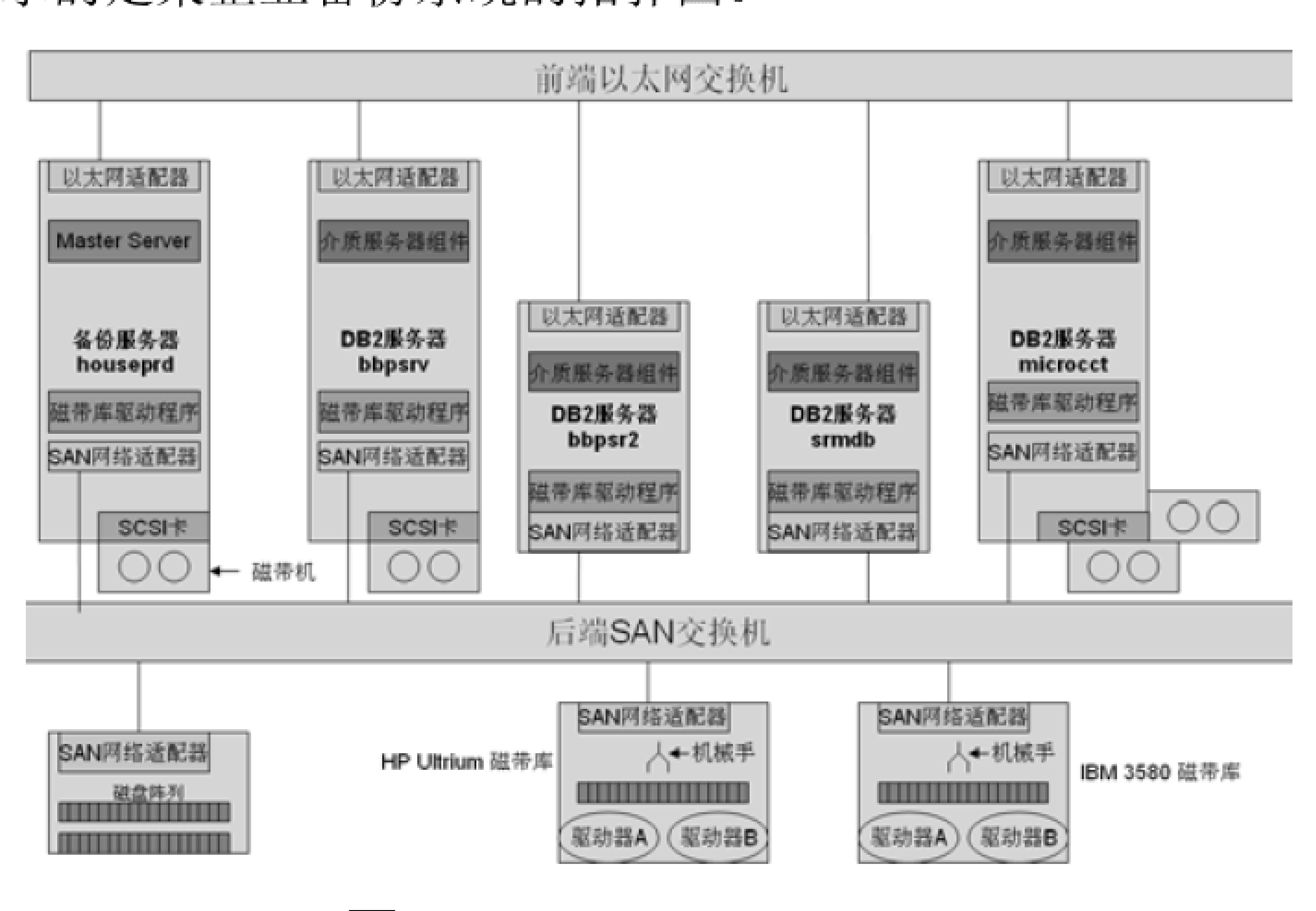


图16.12 某公司 IT 系统示意图

该企业有四台 DB2 数据库服务器，主机名分别为：bbpsrv、bbpsr2、srmdb、microcct，

皆使用 Windows 2000 Advanced Server 操作系统。

其上分别安装 Netbackup 软件的介质服务器模块和磁带库驱动程序。

其中 bbpsrv 服务器连接有一台 SCSI 磁带机，microcct 服务器连接有两台 SCSI 磁带机。这四台数据库服务器也是待备份的服务器。

一台备份服务器，主机名为 houseprd，使用 Windows 2000 Advanced Srever 操作系统，其上安装有 NetBackup 软件的 Master Server 模块(默认包含了 Media Server 模块)，同时也安装了磁带库驱动程序，并且通过 SCSI 线缆连接有一台磁带机。

一台 HP Ultrium 磁带库，包含一个机械手和两个驱动器。

一台 IBM 3580 磁带库，包含一个机械手和两个驱动器。

在这个备份系统中，由 Master Server 进行调度，每台 DB2 数据库服务器都将待备份的数据，通过 SAN 交换机传输给磁带库。由于共有五台服务器使用两台磁带库，所以需要由这五台服务器共享这两台磁带库，为了避免冲突，调度工作统统由 Master Server 来进行。

16.3.7 NetBackup 配置指南

图 16.13 是在 houseped 这台 Master Server 上运行的 NetBackup 配置工具的主界面。右侧窗口是各种自动配置向导。初次安装完 Netbackup 软件之后，首先要让 Netbackup 这个调度员识别到网络上的每台介质服务器，及与其挂接的各种用于备份的存储设备。

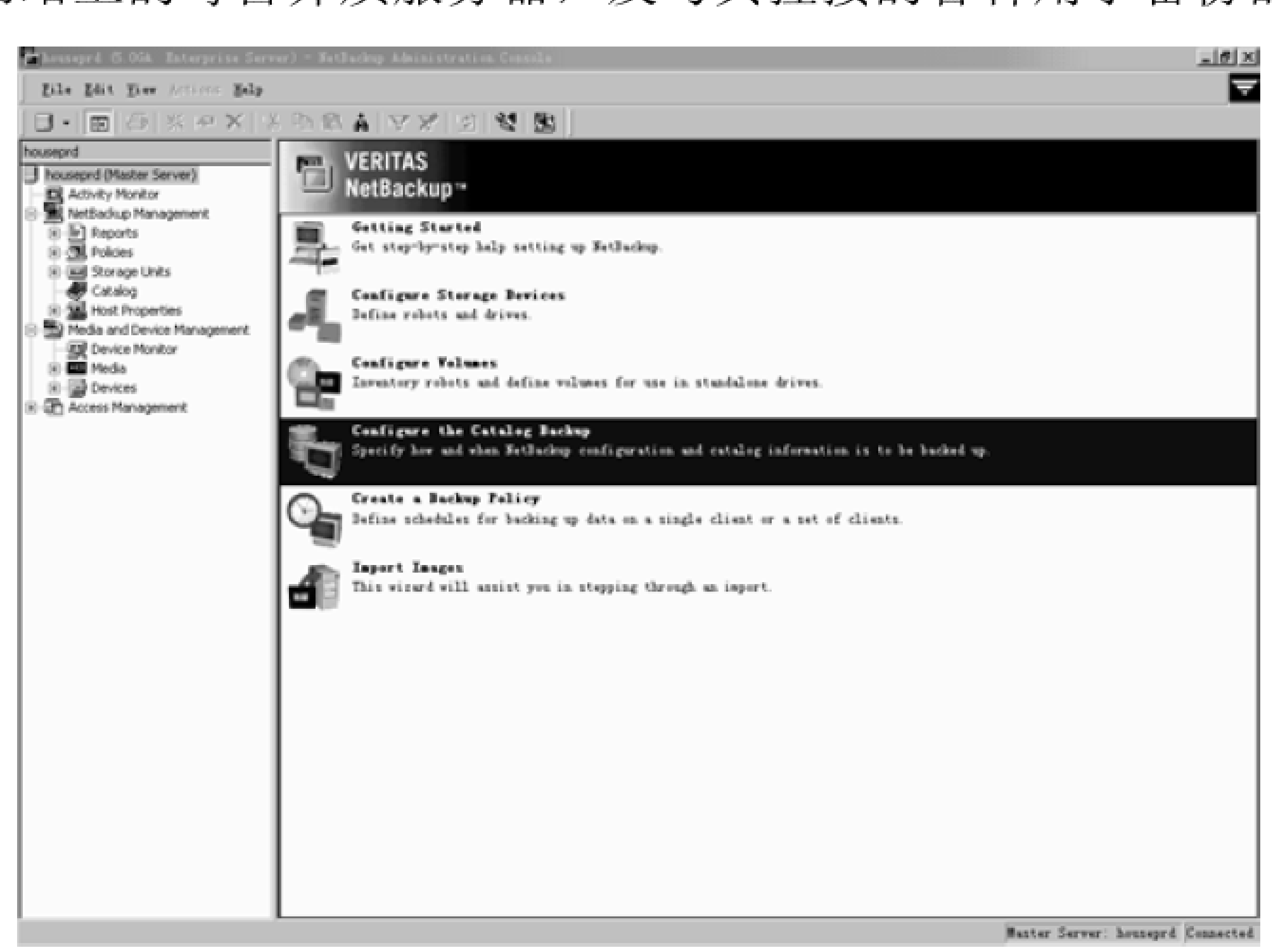


图16.13　NetBackup 主界面

1. 配置存储设备

初次运行 NetBackup 配置工具的时候，可以通过右侧窗口的向导“Getting Started”来让 Master Server 扫描网络上的介质服务器和其上的磁带库设备，并对扫描到的设备以及磁带做一些配置和记录，形成一个初始化环境。

1】 单击“Getting Started”，出现如图 16.14 所示的对话框。

2】 单击“下一步”按钮，出现如图 16.15 所示的对话框。

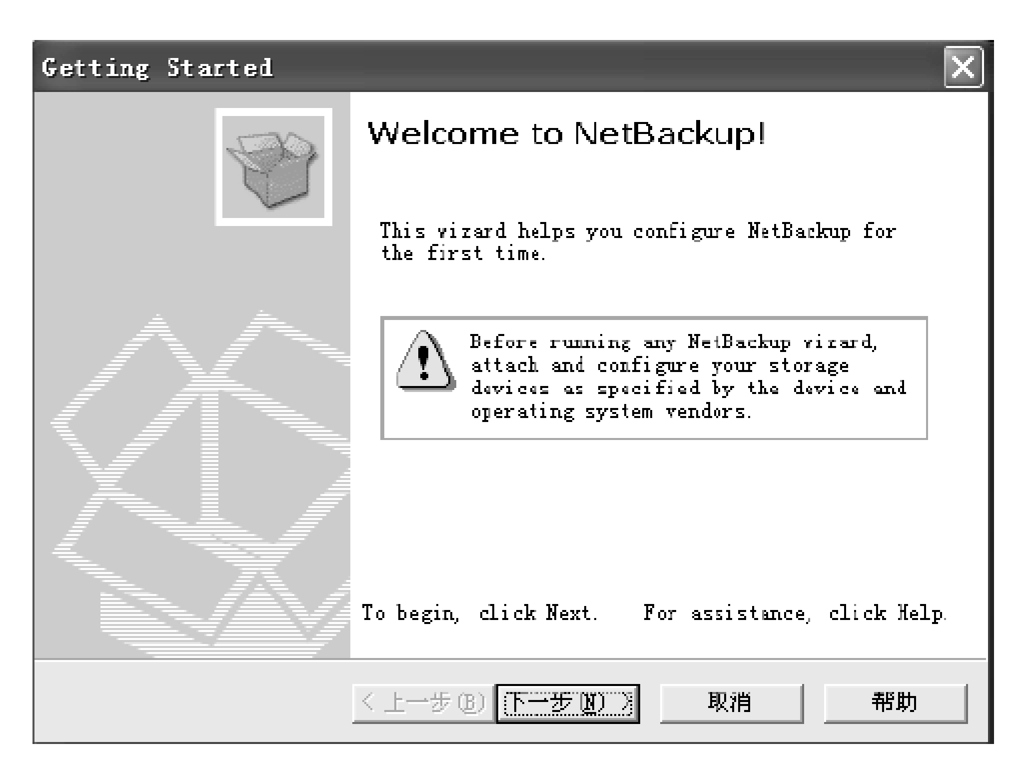


图16.14 Getting Started

图16.15 Device Configuration

3】 对话框中提示，初始化过程需要 4 个步骤，首先扫描所有网络介质服务器可供备份用的设备。单击“下一步”按钮，出现图 16.16 所示的对话框。

4】 单击“下一步”按钮，如图 16.17 所示，扫描到一个介质服务器。



图16.16 扫描介质设备



图16.17 扫描到一个介质服务器

5】 在图 16.17 的对话框中，可以选择网络上的所有介质服务器，这样，就可以扫描这些服务器上用于备份的存储设备了，如图 16.18 所示。

6】 扫描结束后，单击“下一步”按钮，出现如图 16.19 所示的对话框。



图16.18 扫描到介质服务器上的设备



图16.19 设备列表

7】 在对话框中会列出所有检测到的设备的状态，但是图 16.19 中没有找到任何设备，这可能是由于主机未联入网络或者介质服务器没有安装 Media Server 模块。单击“下一步”按钮，出现如图 16.20 所示对话框。

该对话框可用来建立一个硬盘目录，这个目录可供备份文件存放，但不是必须的。一般将备份后的数据存入磁带中，用磁盘目录存放备份数据的一个好处就是可以作为一个缓冲，可以设置 NetBackup 在一定时间后，将这个硬盘上的数据转移到其他备份目的中。

8】 单击“下一步”按钮，如图 16.21 所示。

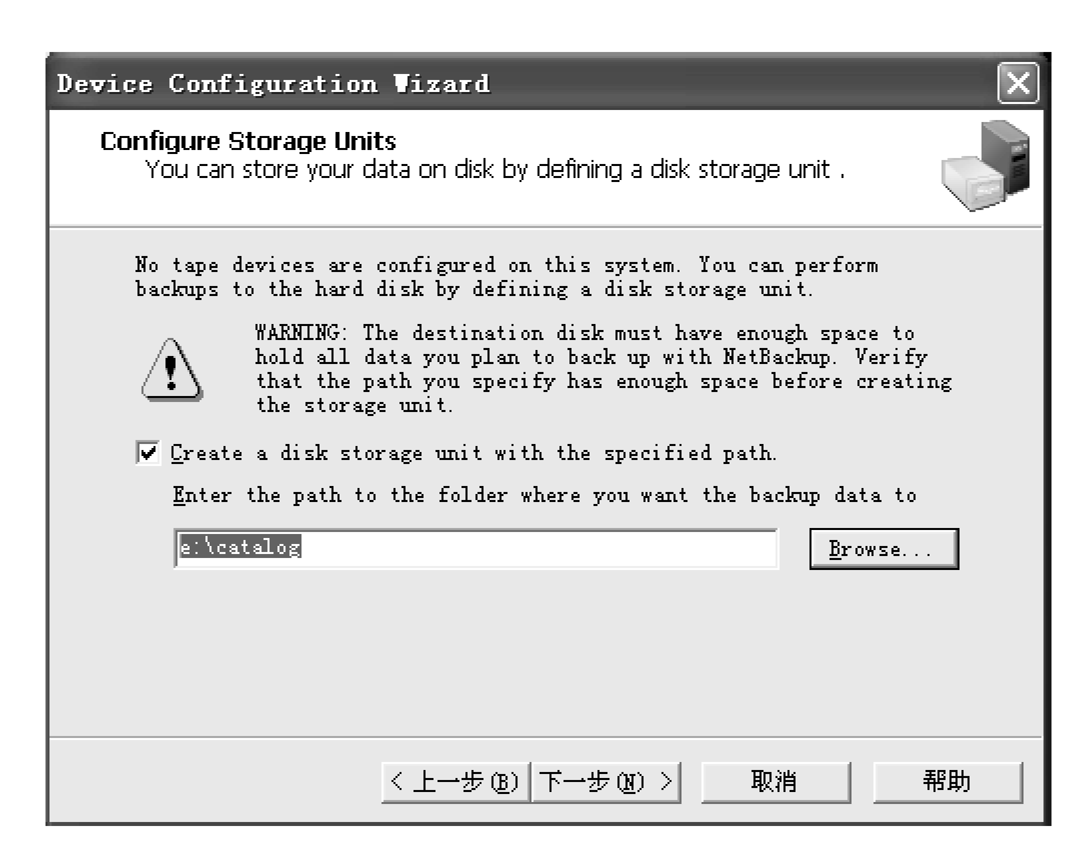


图16.20　建立本地磁盘备份目录



图16.21　设置完成

9】 单击“完成”按钮。将进入 Volume 配置界面，如图 16.22 所示。

10】 所谓 Volume，指的就是磁带在后面会详细介绍。这一步中，NetBackup 会识别所有磁带库中的磁带，并将它们编入默认的 Volume Group 中供使用。单击“下一步”按钮继续，如图 16.23 所示。

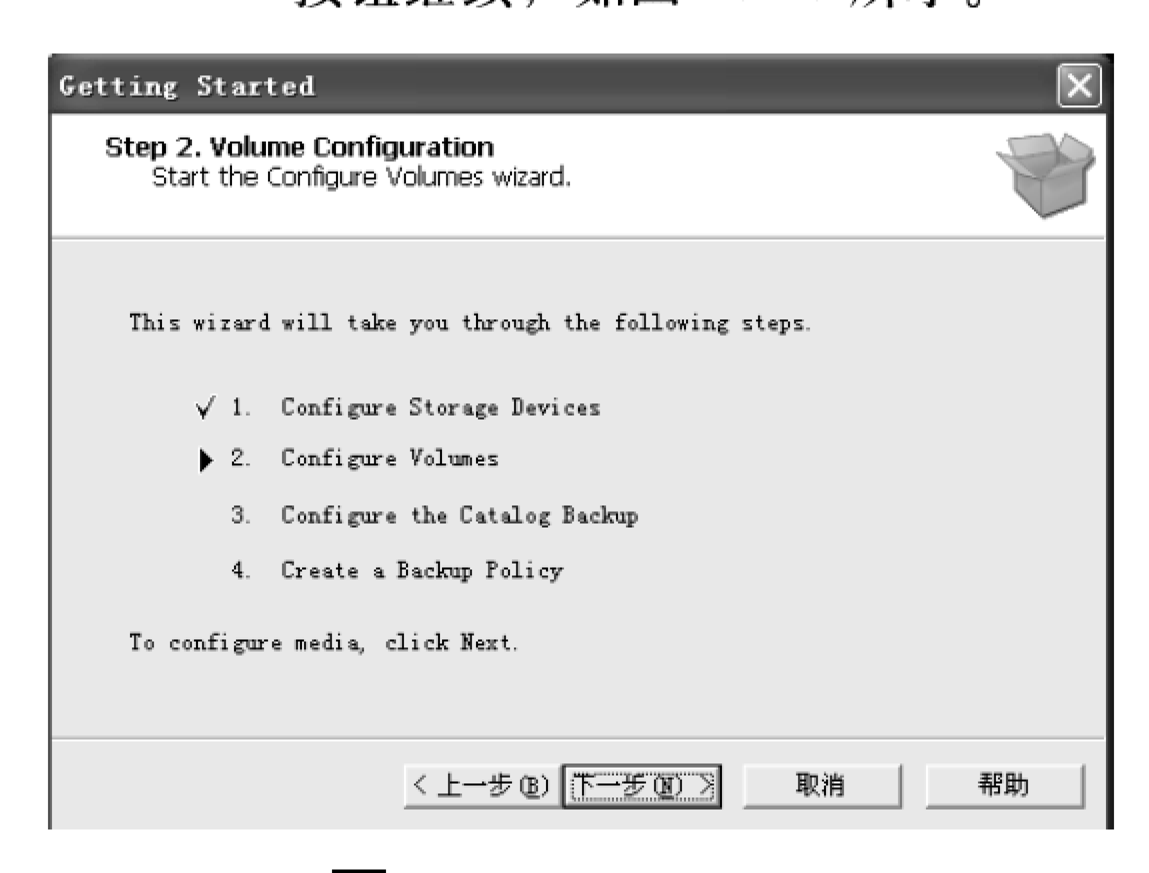


图16.22　配置 Volume

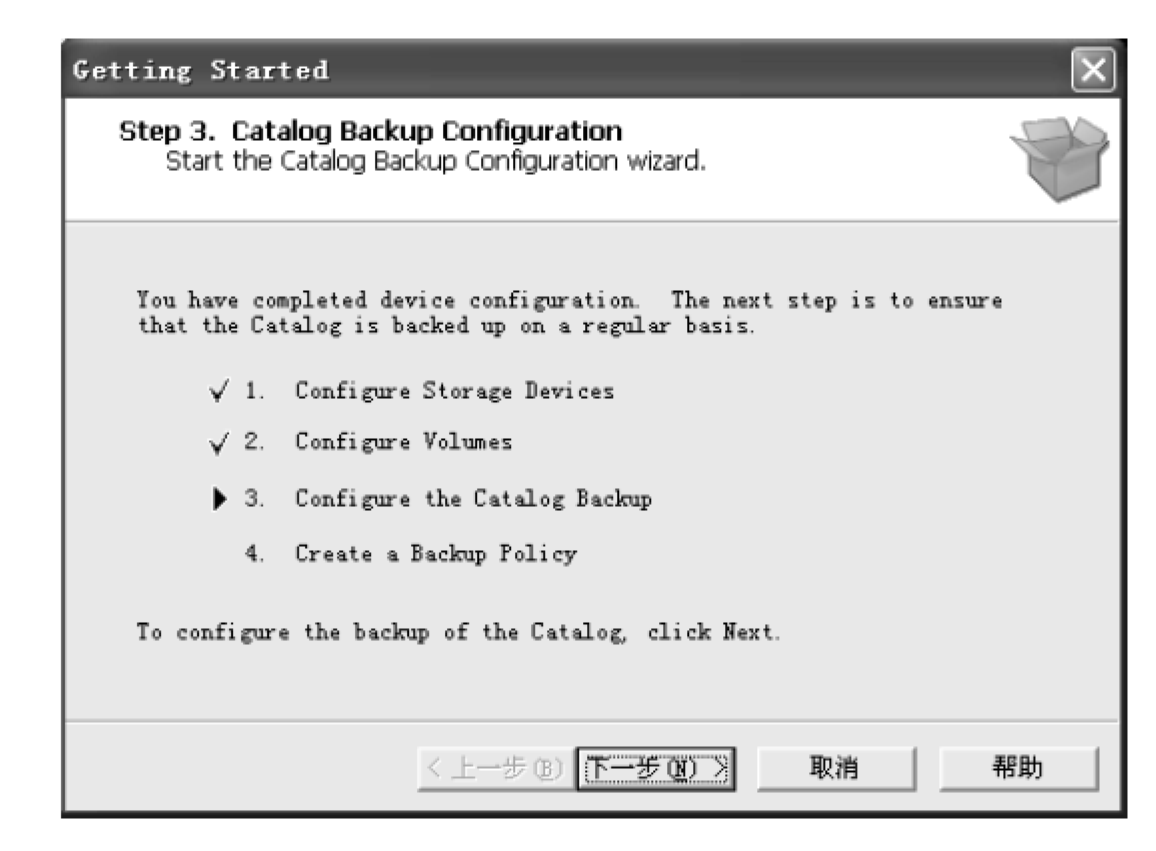


图16.23　设置 Catalog 备份

11】 设置 Catalog 的备份。所谓 Catalog，就是 NetBackup 自身运行所需要的数据，其实 NetBackup 自身管理维护着一个小型数据库，数据库中保存了 NetBackup 的所有配置，以及所有磁带、设备、备份策略、过期时间等信息，如果 Catalog 损坏，则整个 NetBackup 将会瘫痪，所以备份 Catalog 自身也是非常重要的。这就像医生

给别人治病的同时，自己也要预防疾病一样。NetBackup 虽然是一个备份其他数据的软件，但是它也要备份好自身的数据，这一点很好理解。单击“下一步”按钮，如图 16.24 所示。

12】 单击“下一步”按钮，如图 16.25 所示。

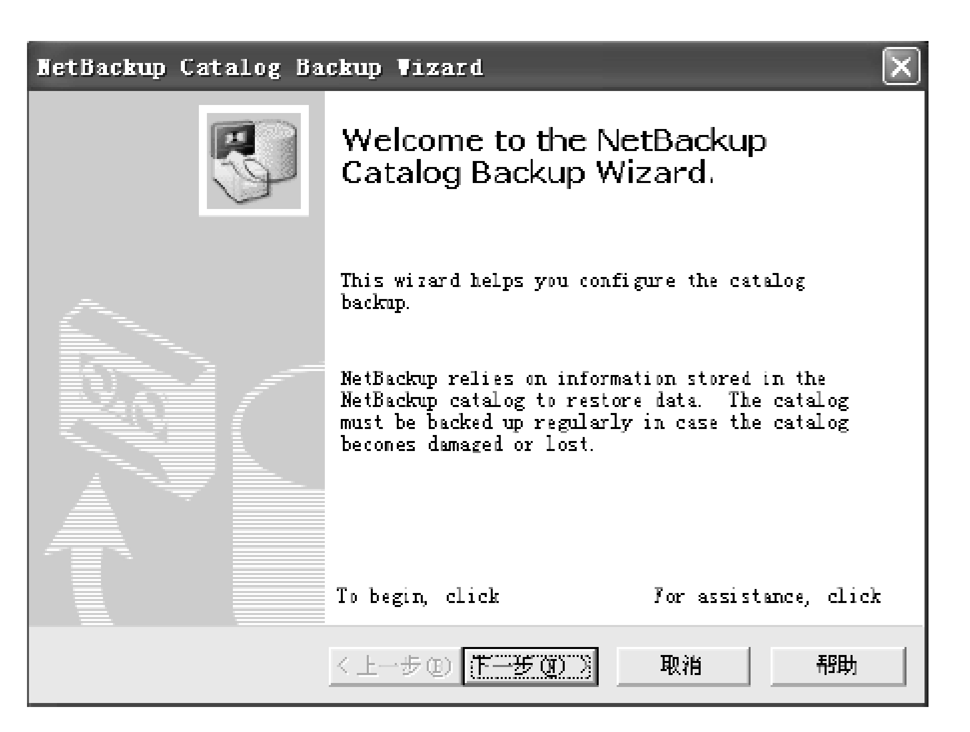


图16.24 Catalog 备份向导

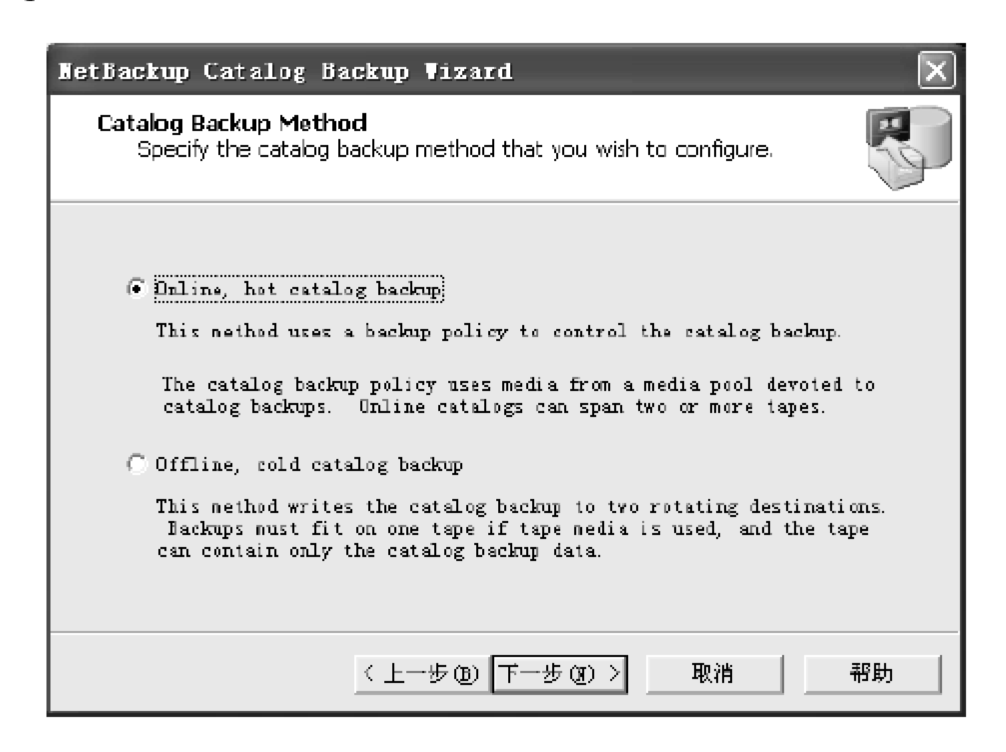


图16.25 选择 Catalog 备份方式

13】 选择“Online，hot catalog backup”单选按钮，单击“下一步”按钮，如图 16.26 所示。

14】 创建一个用于备份 Catalog 信息的新策略，单击“下一步”按钮，如图 16.27 所示。

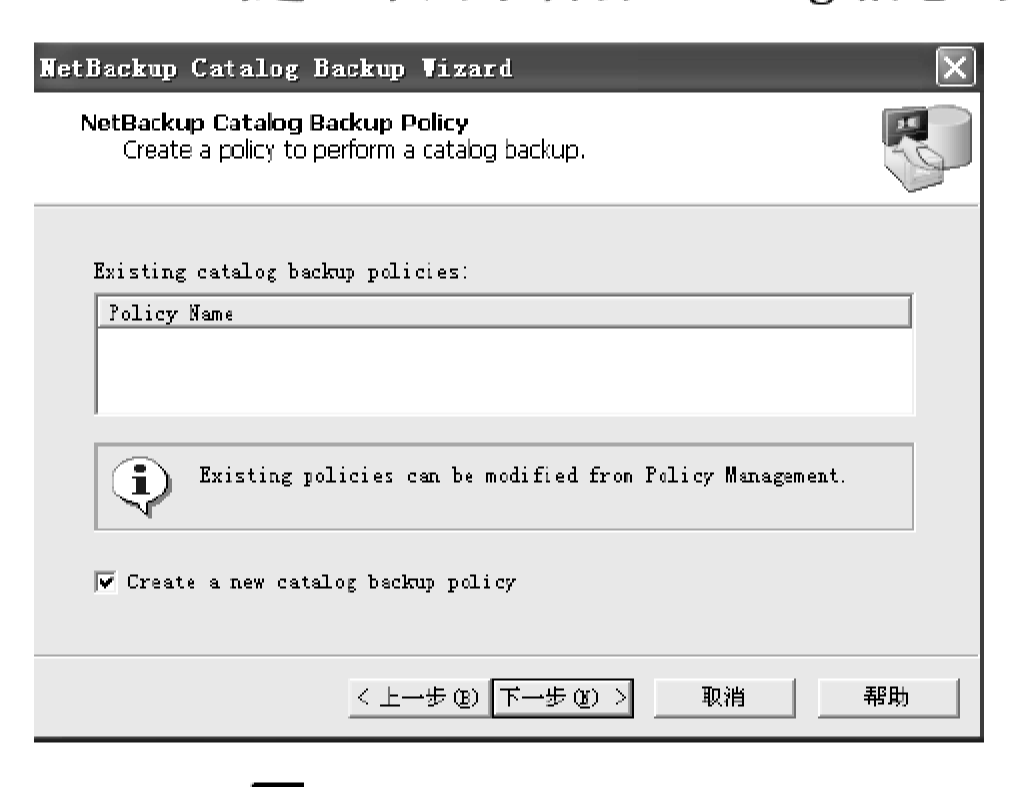


图16.26 创建备份策略

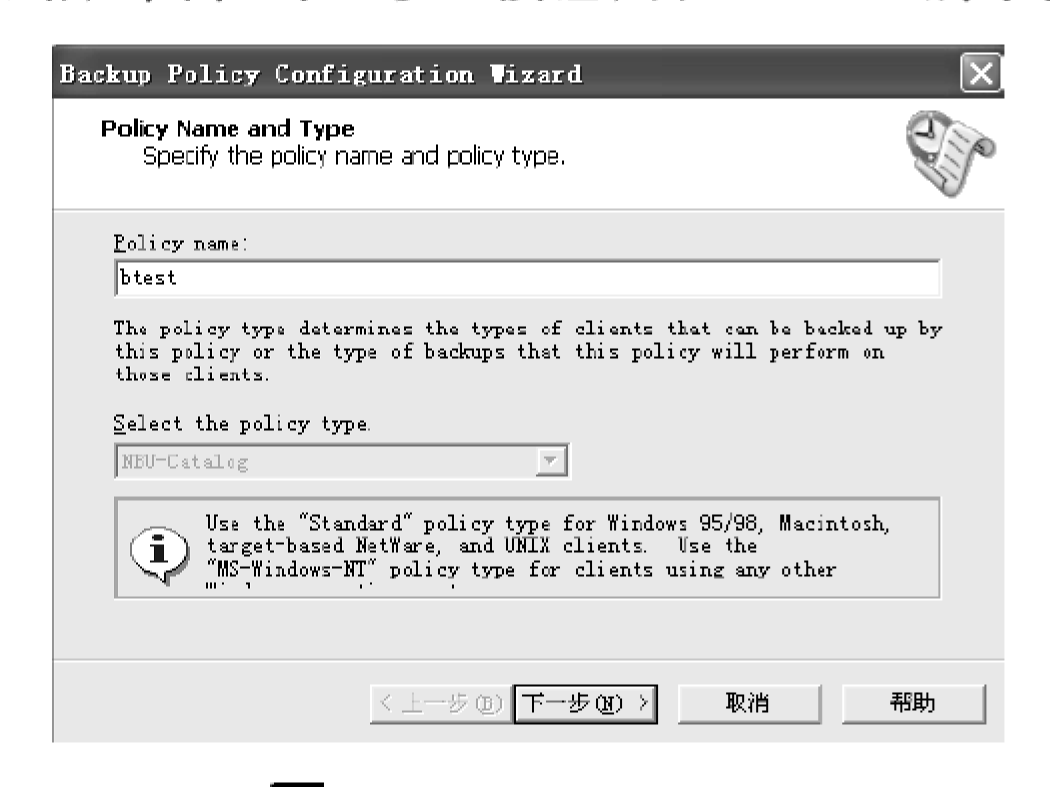


图16.27 输入策略名称

15】 输入 btest，单击“下一步”按钮，进入备份方式选择窗口，如图 16.28 所示。

16】 选择完全备份或者增量、差量备份。单击“下一步”按钮继续，如图 16.29 所示。



图16.28 全备和增备方式选择

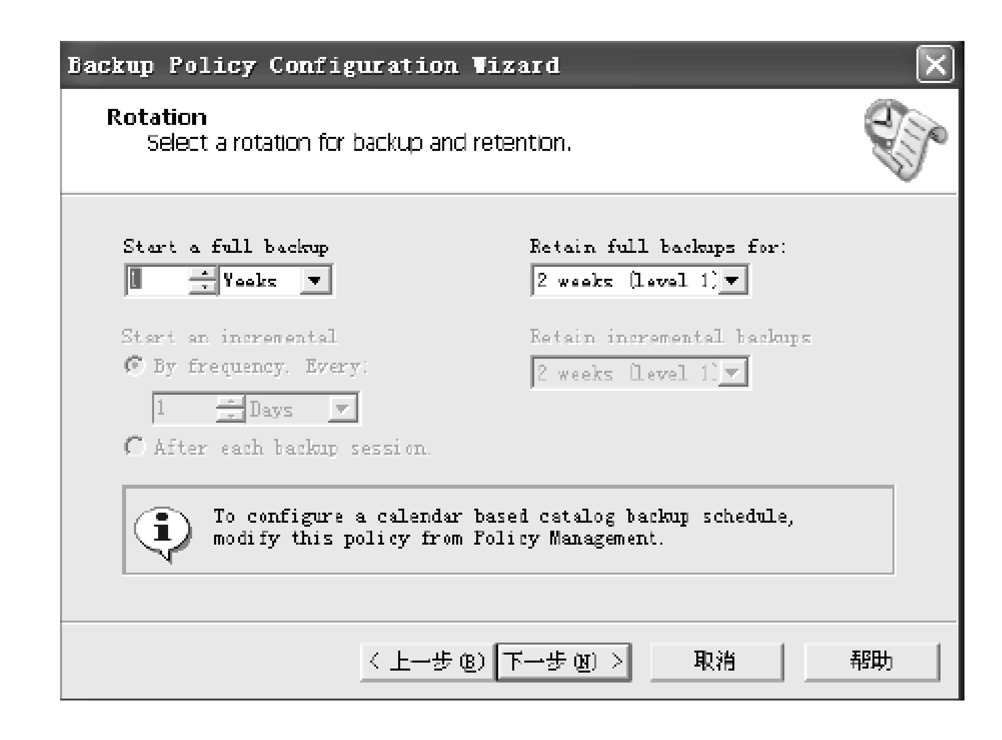


图16.29 设置 Catalog 备份日程表

17】 设置每周进行一次完全备份，每个备份保留期限为 2 周，2 周过后，之前的备份就认为失效，存放备份的磁带可供其他备份使用，如图 16.30 所示。

18】 选择具体备份时间，图 16.30 跨越了所有时间，所以备份可以在任何时间内发生。继续。

19】 设置备份后的 Catalog 信息存放位置。以及登录操作系统所需的用户认证信息。单击“下一步”按钮继续，如图 16.31 所示。

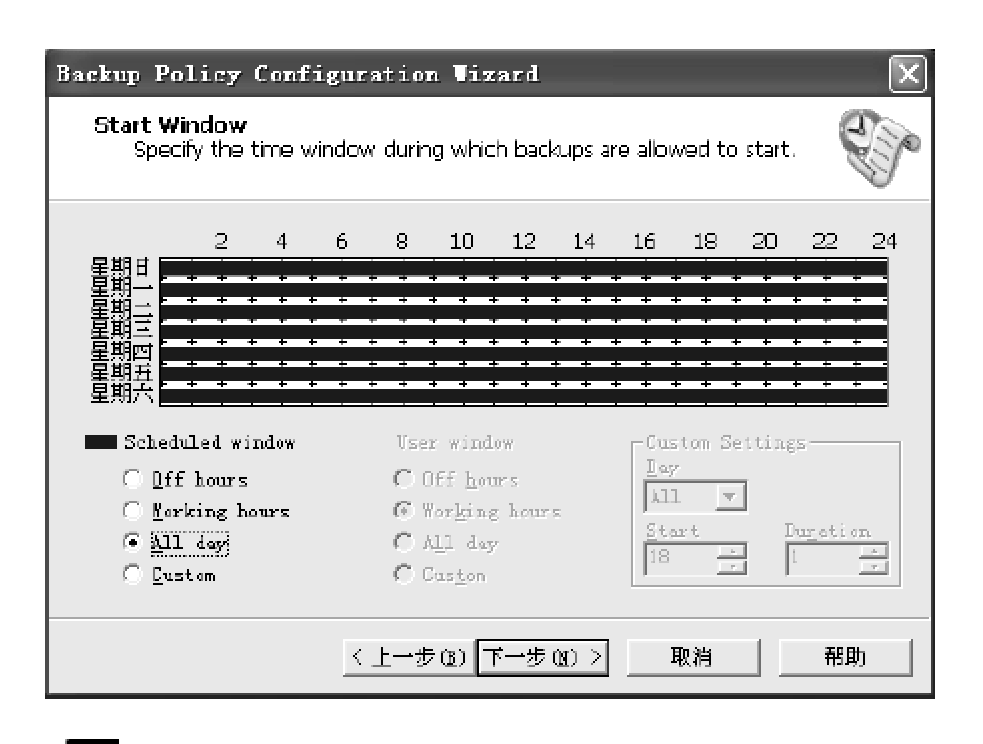


图16.30　选择具体 Catalog 备份日程表

Backup Policy Configuration Wizard
Catalog Disaster Recovery File
Specify a file location to save the disaster recovery information
Path:
c:/btest
Browse
Logon:
veritas
Password:
12345679
The disaster recovery file generated for each catalog backup contains information needed to recover the NetBackup catalog.
Record the location of this file so that the NetBackup catalog can be recovered if necessary.

图16.31　选择 Catalog 备份路径及认证信息

20】 设置是否进行邮件通知，如图 16-32 所示。选择 No 单选按钮，单击“下一步”按钮继续，如图 16.33 所示。

21】 完成 Catalog 备份策略向导。单击“下一步”按钮，如图 16.34 所示。

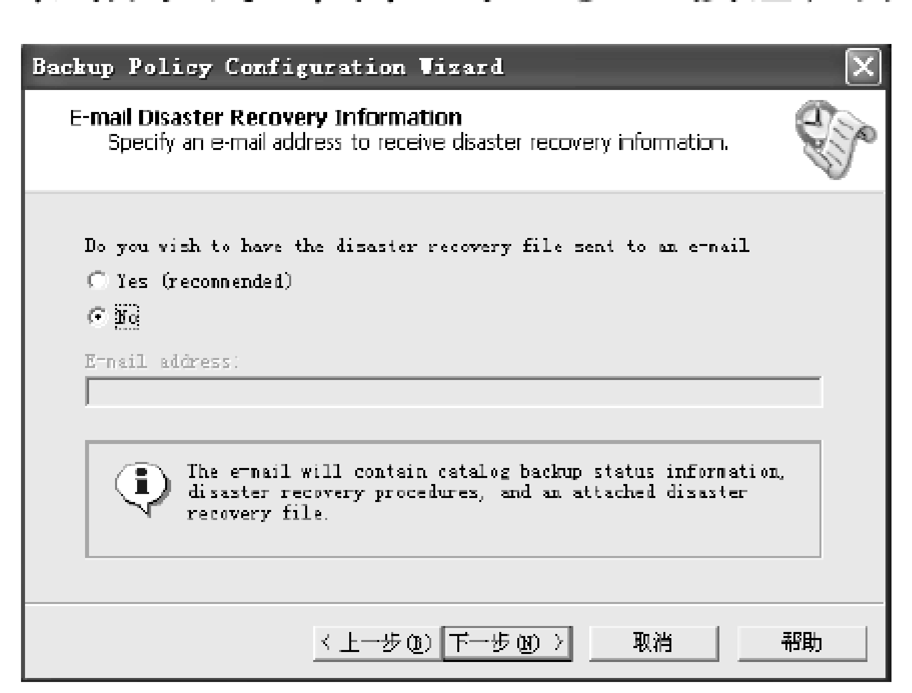


图16.32　是否邮件通知



图16.33　完成设置

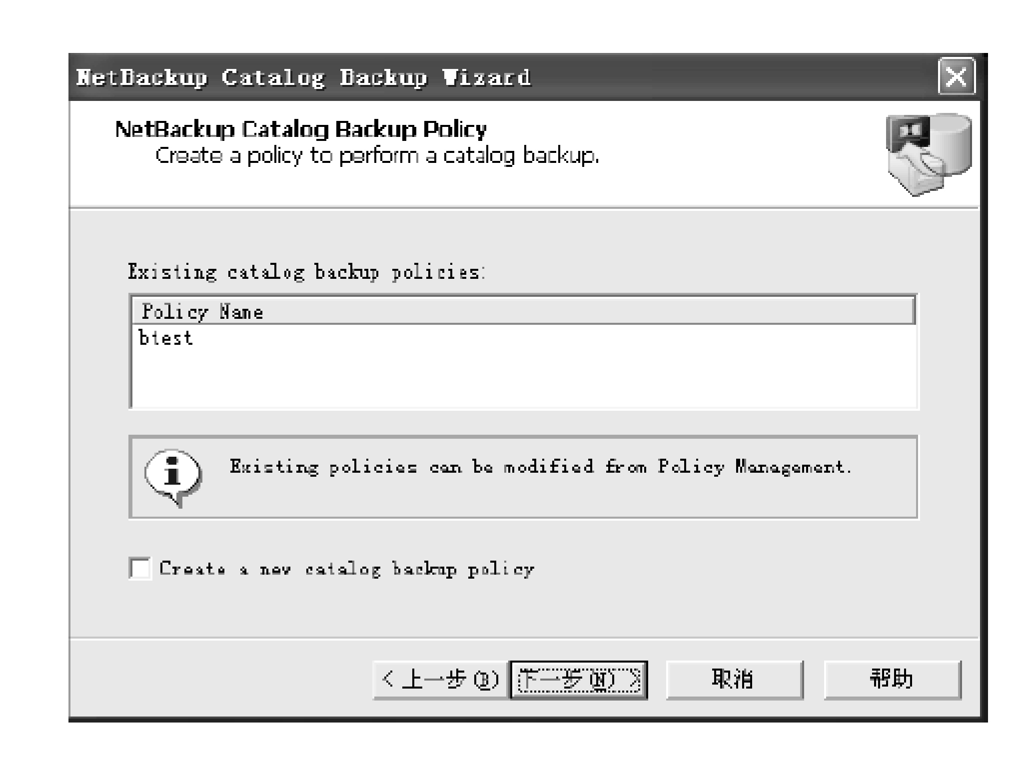


图16.34　策略列表

22】 选择新创建的策略，单击“下一步”按钮，即可完成环境的初始化操作。

初始化后，Master Server 会在 Media Server 列表中自动加入这些扫描到的介质服务器，并且在 Storage Unit 中列出扫描到的机械手设备。图 16.35 所示的是介质服务器列表，图 16.36 所示的是备份客户端列表。

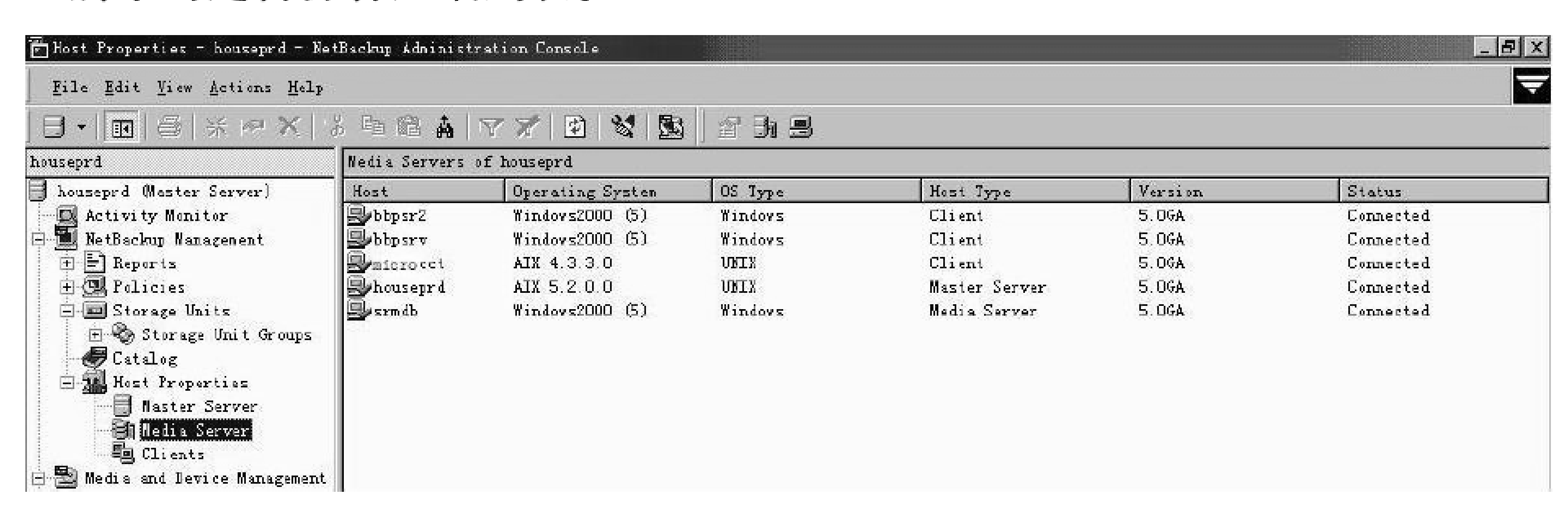


图16.35 介质服务器列表

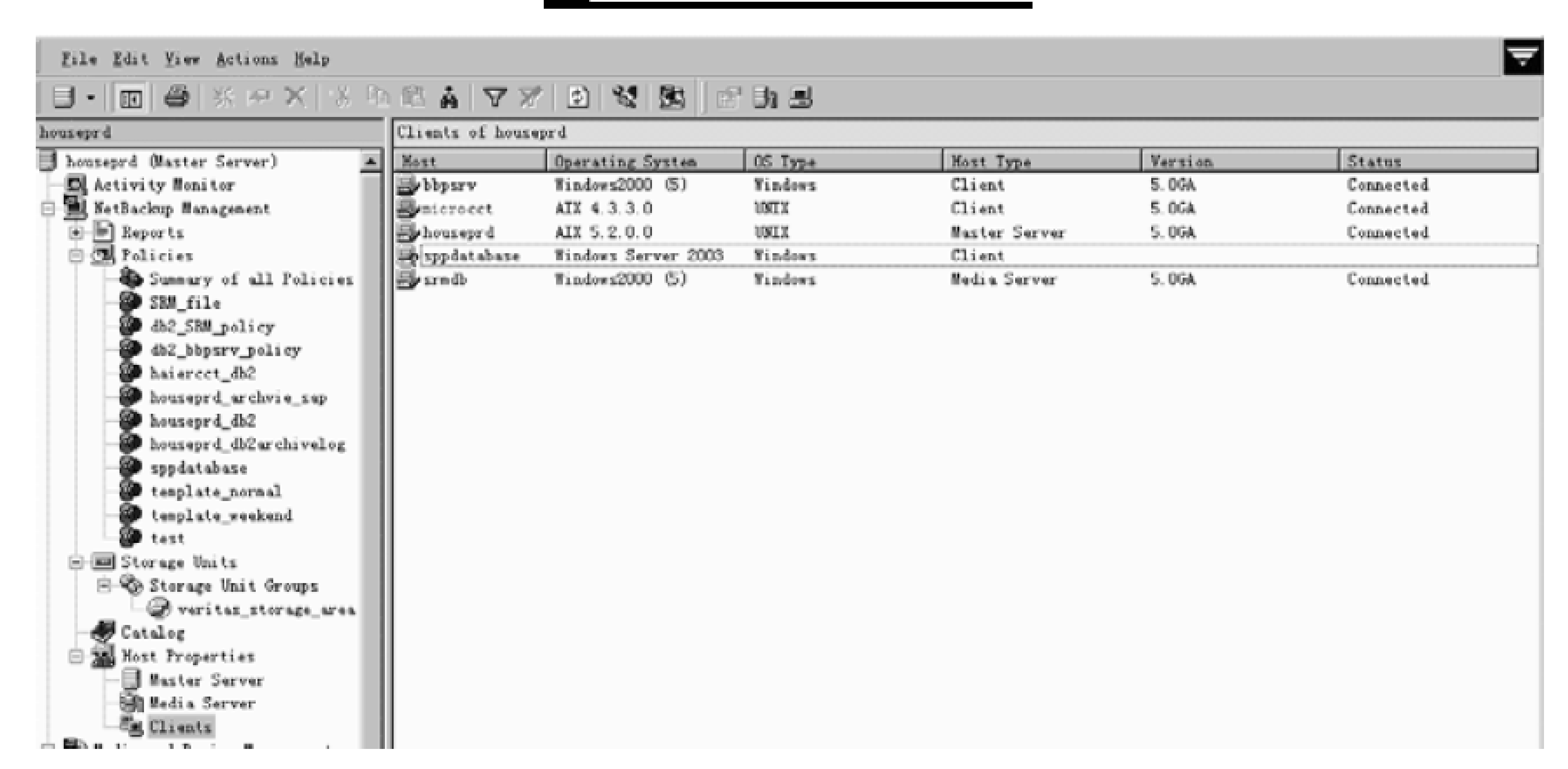


图16.36 备份客户端列表

图 16.37 所示的是存储单元列表。

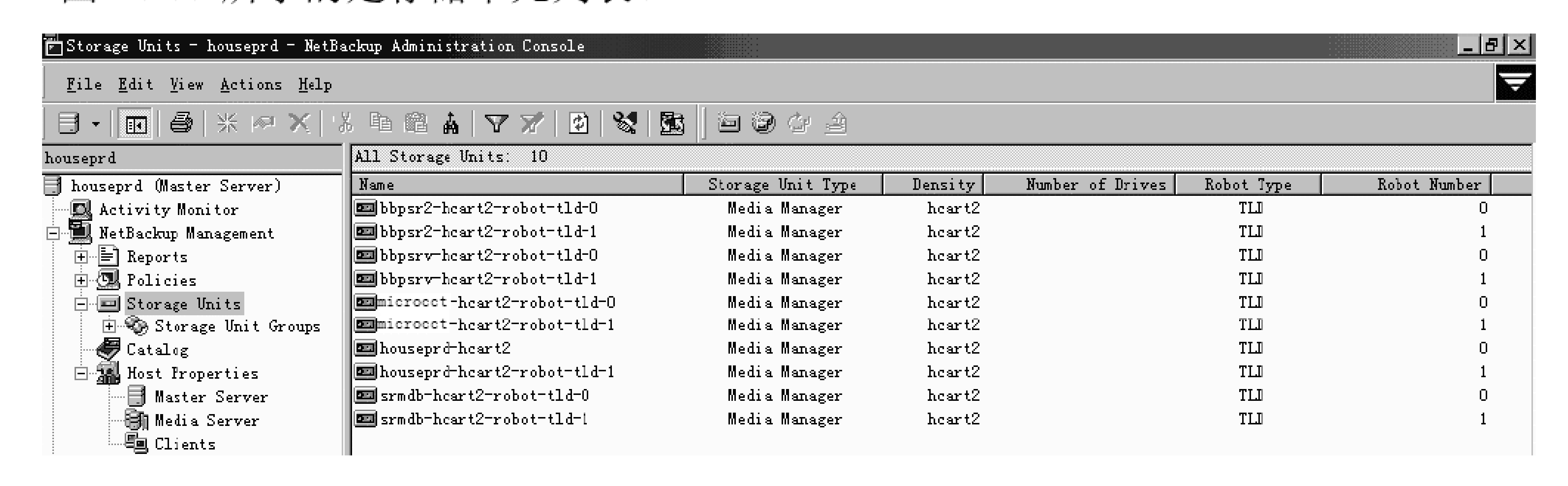


图16.37 存储单元列表

2. Storage Unit

Storage Unit(存储单元)，是一个逻辑上的概念。它表示存储设备中管理一组介质的单元，对于磁带库设备来说，一个机械手就可以掌管属于它的所有磁带，那么一个机械手就是一个存储单元。所以图 16.37 中，每个磁带库的机械手，都被认为是一个存储单元。可

以看到右侧窗口中显示了 10 个机械手设备，但是物理上只存在两个，这是为何呢？

因为五台服务器共享两个机械手，每台服务器都会识别到两台磁带库的机械手，所以一共是 10 个机械手设备。实际使用的时候，只允许其中两台服务器同时操纵两台磁带库设备。但是一段时间内，五台服务器均有机会操纵磁带库，这也就是共享磁带库的意义了。

每台磁带库中的可用介质(磁带)也会被自动添加到 Media 项，如图 16.38 所示。

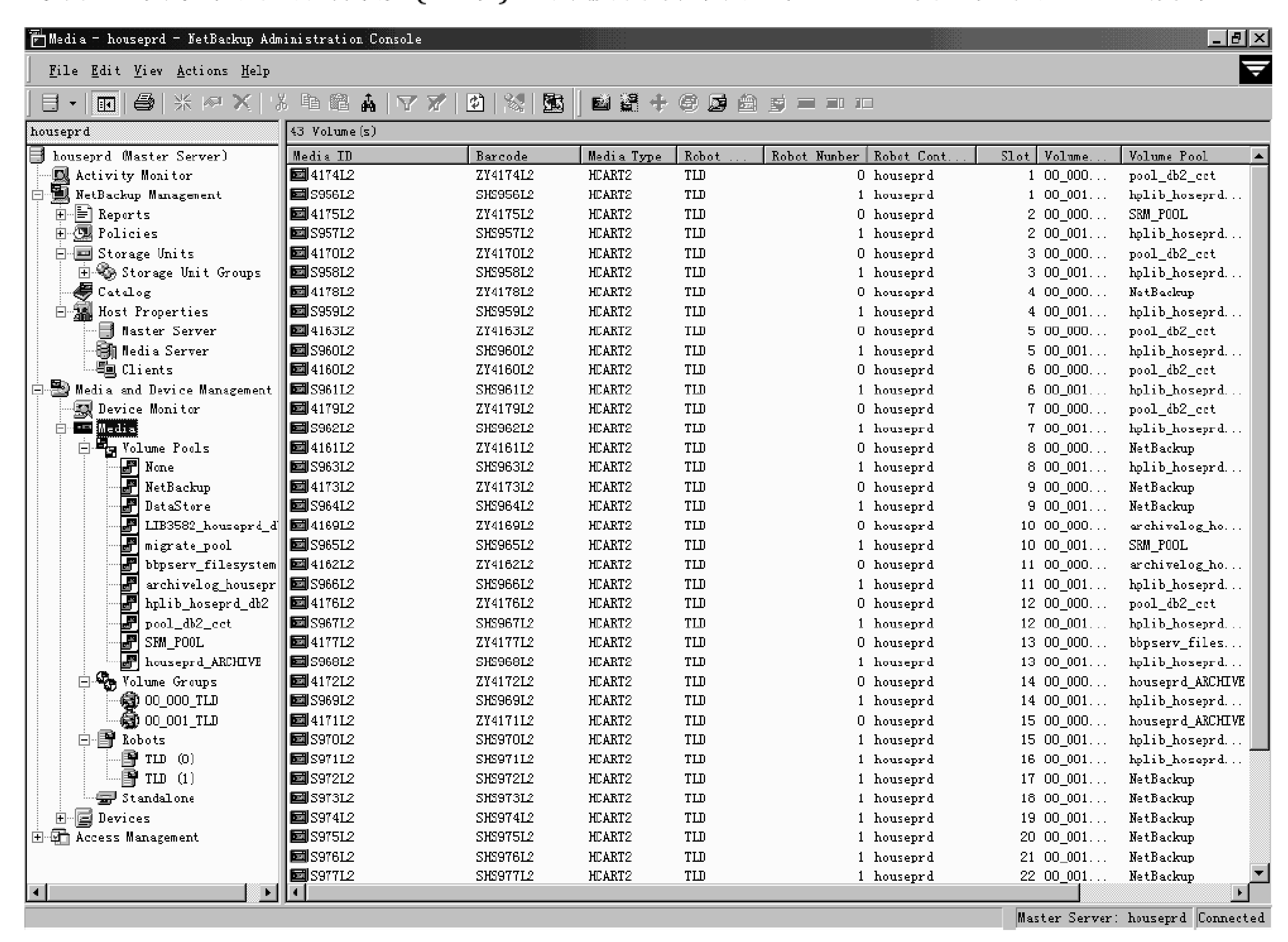


图16.38　介质(磁带)列表

图 16.38 中的 Robots，表示物理上存在的磁带库的机械手，所以只有两个。右侧窗口所示的是识别到的所有磁带，每盘磁带都被编了号，以便加以区分。实际上，每盘磁带都会贴有一个条码，机械手扫描这个条码以区分每盘磁带。

3. 卷池(Media Pool)

由于每盘磁带的存储容量有限，如果有备份需要用到多于一盘磁带，则如何分配并在分配后记录这些磁带的使用状况，是个比较麻烦的问题。为了使管理更加方便，NetBackup 引入了卷池(Media Pool)的概念。这就像磁盘阵列设备将每个物理磁盘合并，并再分割成更大的 Volume 或者虚拟磁盘一样，磁带同样可以这样被虚拟化。如图 16.39 和图 16.40 所示。

我们看到，Volume Pools 项之下的 11 个卷池，其中名为 NetBackup 的卷池包含了 11 盘磁带。而名为 bbpserv_filesytem 的卷池，只包含了一盘磁带。有了卷池之后，就可以把卷池中的所有磁带，当成一个大的虚拟磁带来看待。我们可以为每个待备份的数据项目分配一个卷池，每次备份的数据只存放在这个卷池中，其他卷池中的磁带不会给这个备份所使用，这样就做到了充分的资源隔离。卷池可以手动创建并且在不冲突的前提下任意添加磁带，如图 16.41 和图 16.42 所示。

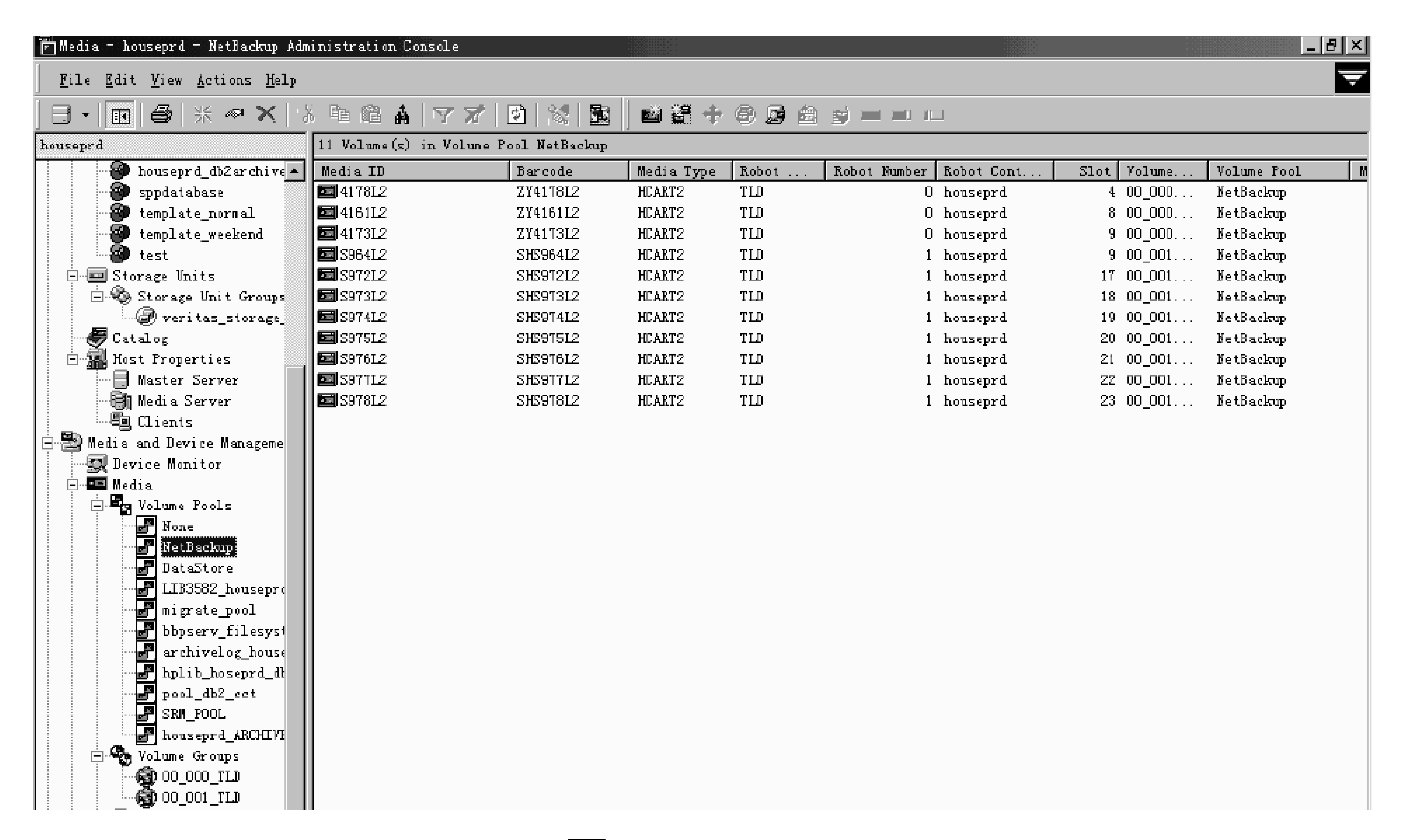


图16.39 卷池(1)

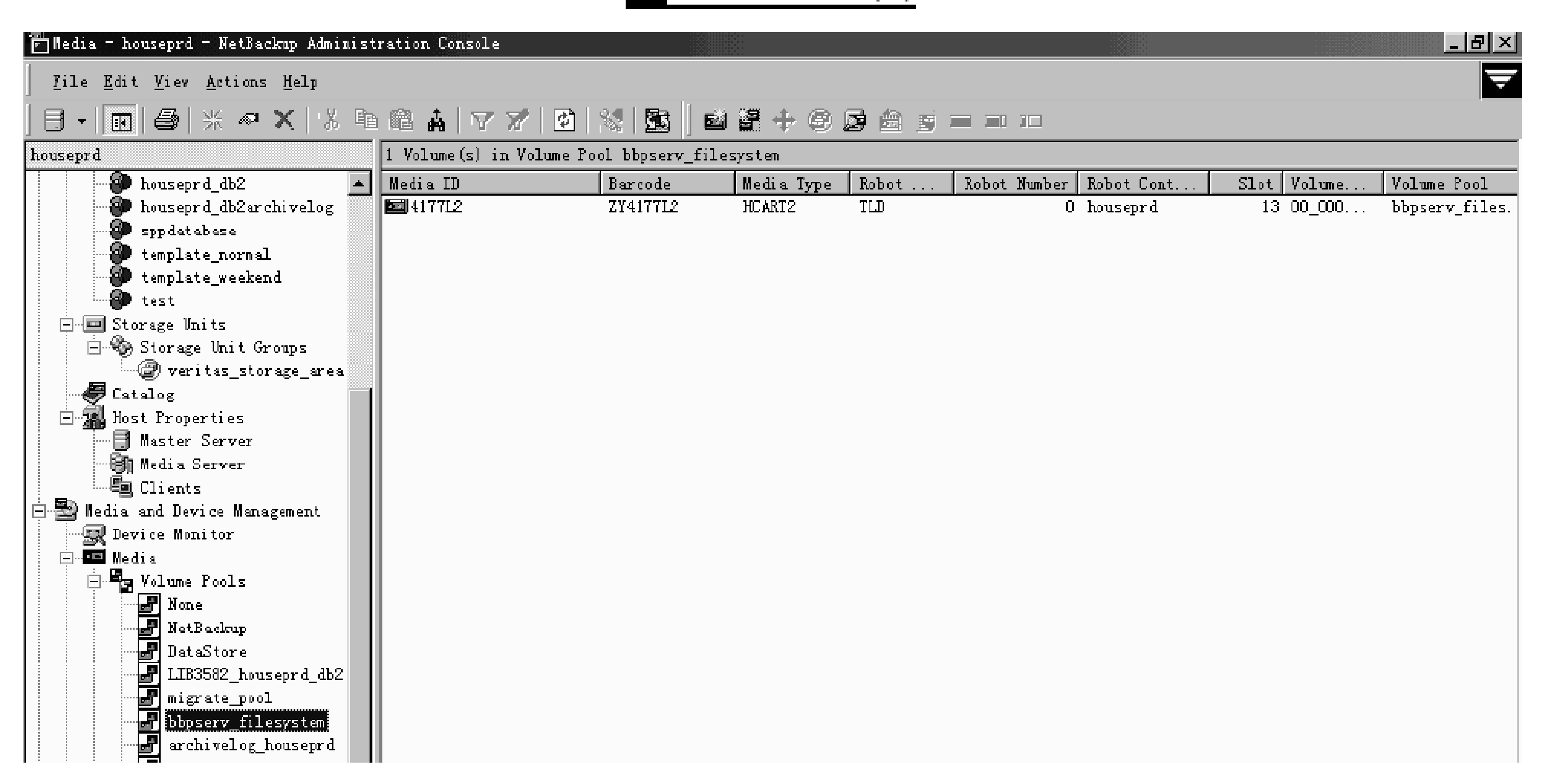


图16.40 卷池(2)

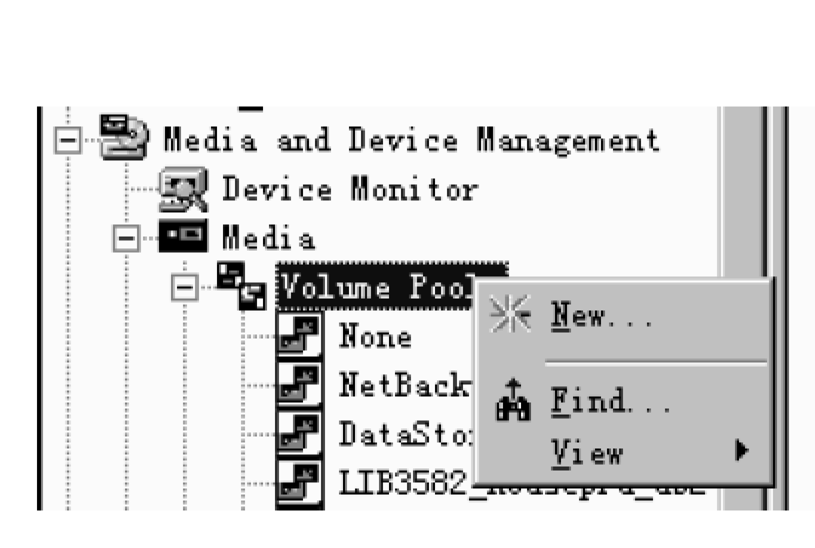


图16.41 新建卷池

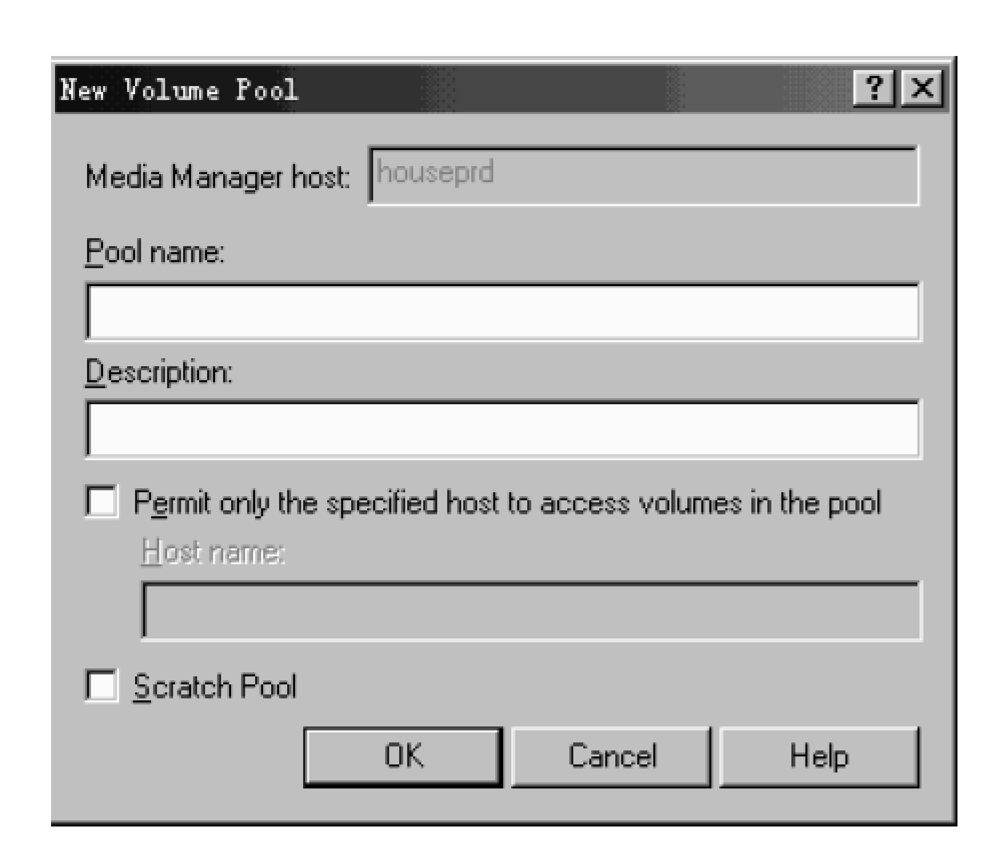


图16.42 输入名称和描述

4. 卷组(Media Group)

这也是一个逻辑上的概念，下图中显示了两个卷组，每个磁带库中的磁带，都放到了一个单独的组中。卷组在实际使用上没有很大的意义。卷组不能手动创建，默认每个机械手就会生成一个卷组，如图 16.43 所示。

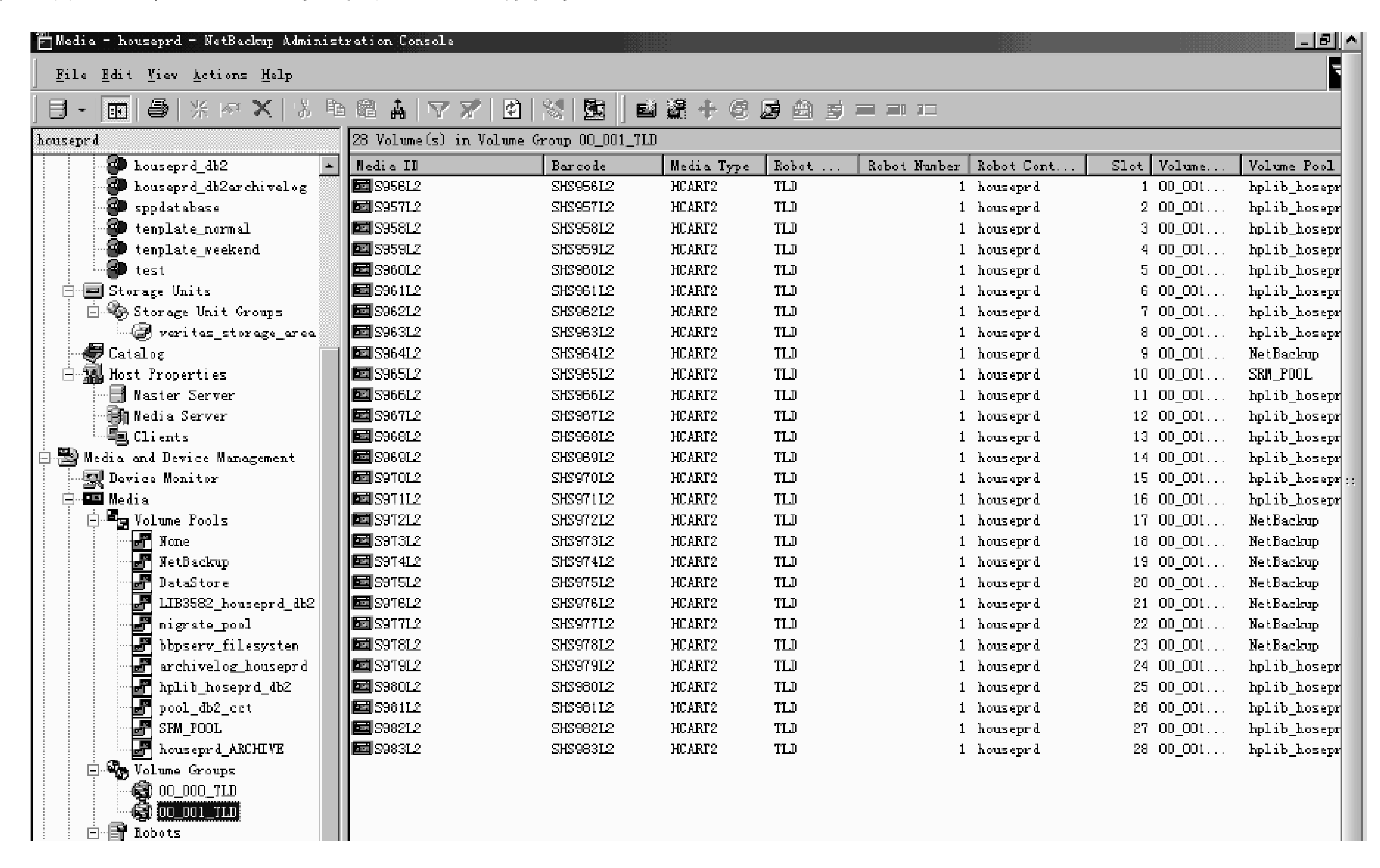


图16.43　卷组

5. Robots(机械手)

如图 16.44 所示，左侧显示的是机械手。

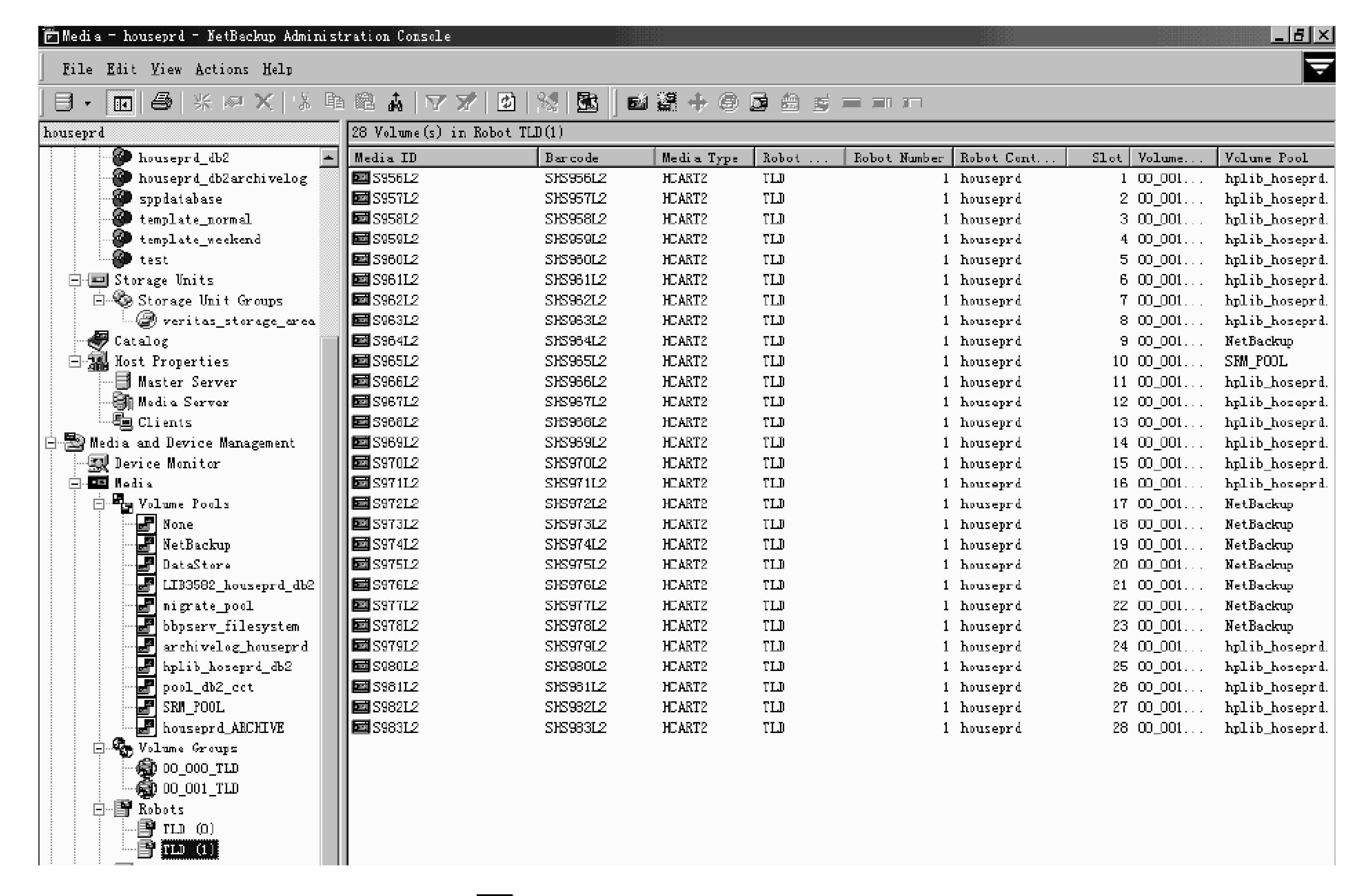


图16.44　左侧显示的机械手

这个项下面显示了整个备份系统中所存在的物理机械手的数量，右侧窗口中显示了对应机械手所掌管的磁带。

6. Standalone(非共享的机械手)

如图 16.45 所示是 Standalone 机械手。

如果某台介质服务器独立掌管一台磁带库的机械手，而没有共享给其他主机使用，则 NetBackup 识别到这种设备之后，就会显示在右侧窗口中。本例中没有这种设备。

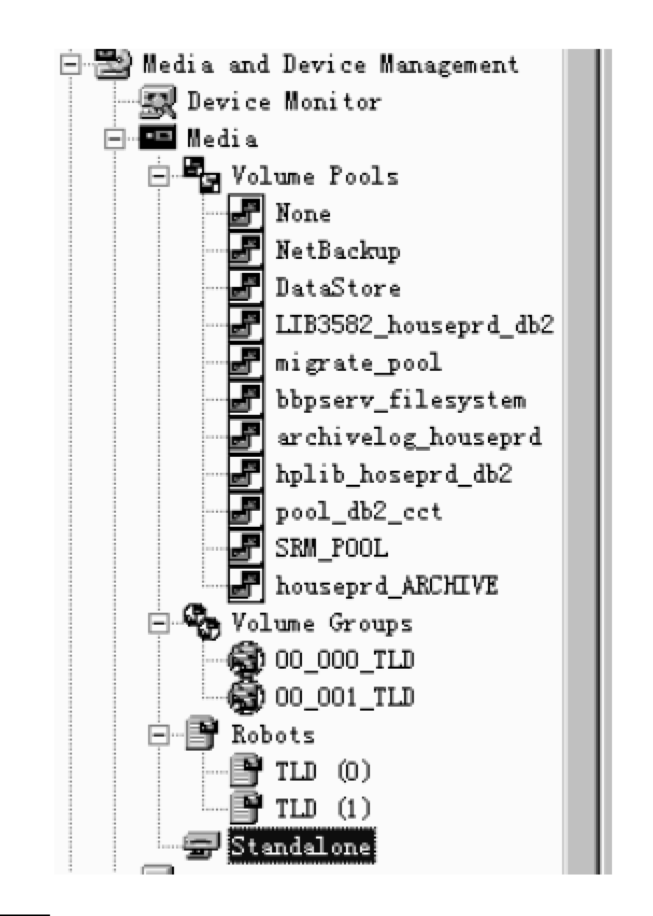


图16.45　Standalone 的机械手

7. Devices(设备)

这一项列出了整个系统中所有可用于备份的物理存储设备，如图 16.46 所示。

右侧窗口中的拓扑图显示了两台磁带库和四台独立磁带机，并有连线。带有齿轮标志的为介质服务器，右上方的图标为磁带库，其中还显示了机械手、磁带槽和两个驱动器，驱动器下面的手托表明这个驱动器为共享驱动器，也就是说其他主机也可以操作这个驱动器。

右侧窗口的下半部分显示了所有逻辑而不是物理设备。由于共享驱动器的原因，本例中的逻辑驱动器变为了 12 个(三台服务器，每台识别到四个驱动器)，再加上独立的磁带机，共有 16 个驱动器，如图 16.47 所示。

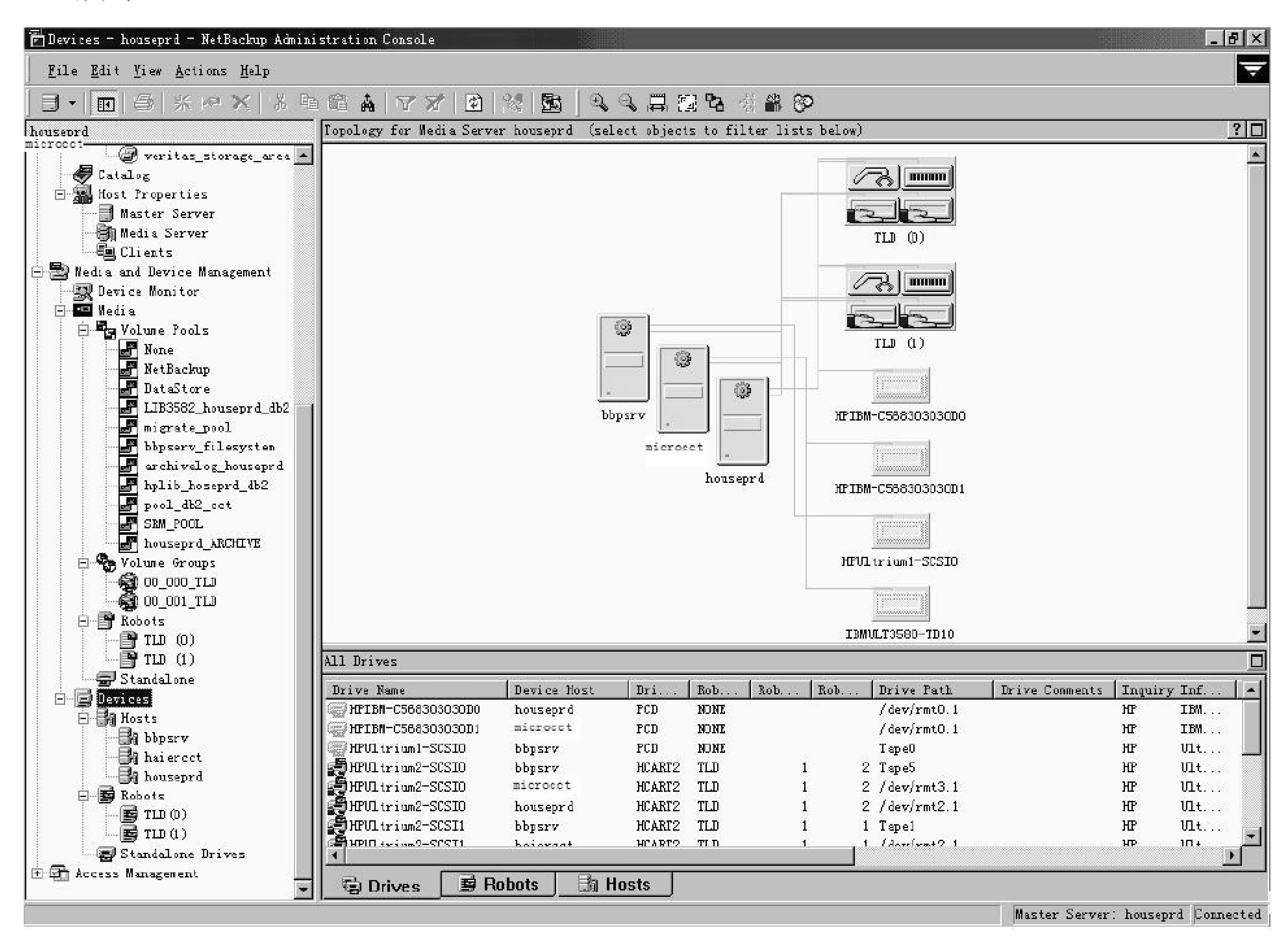


图16.46　所有可用于备份的介质设备

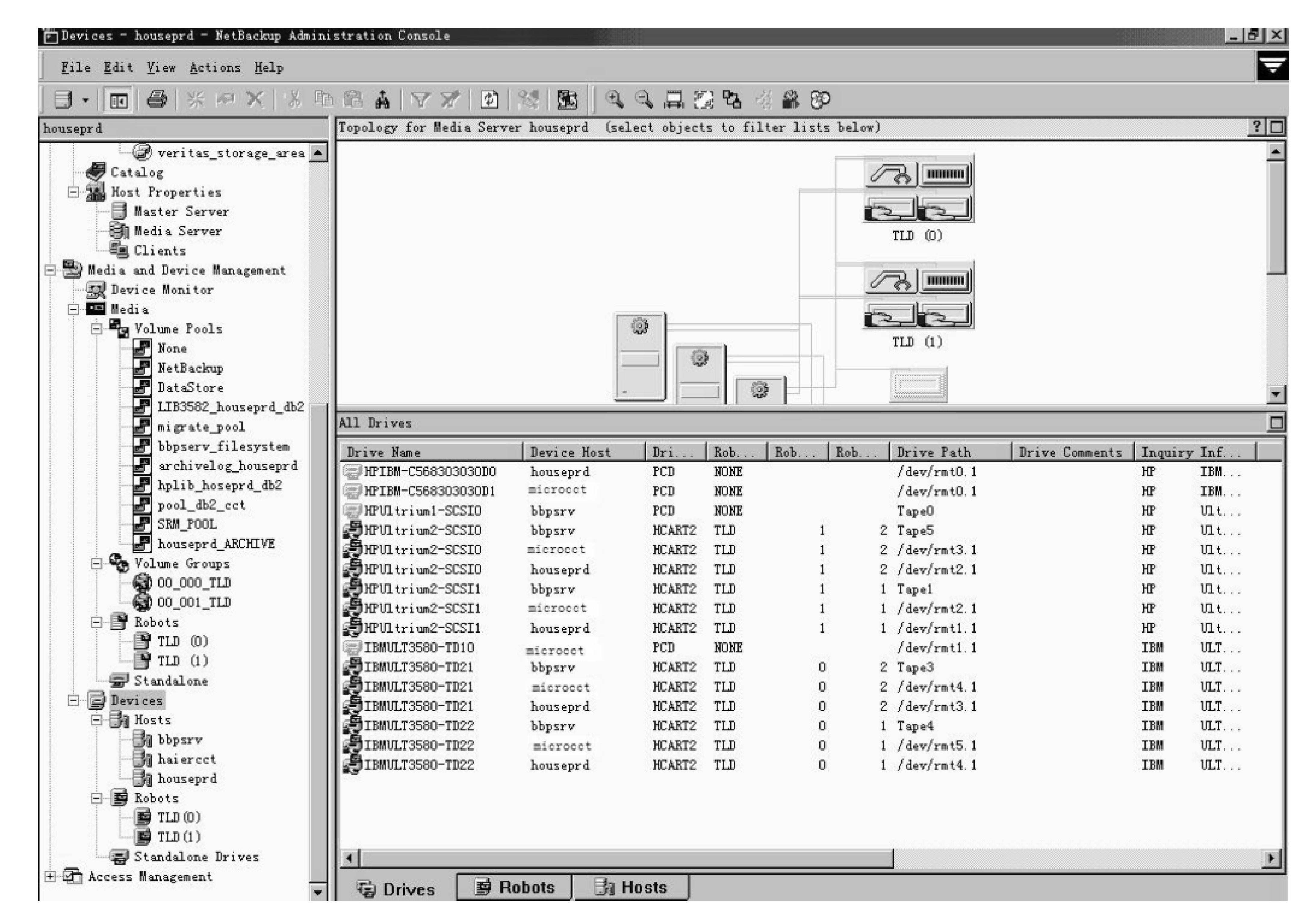


图16.47 设备列表

同理，逻辑机械手也有 6 个而不是 3 个，如图 16.48 所示。

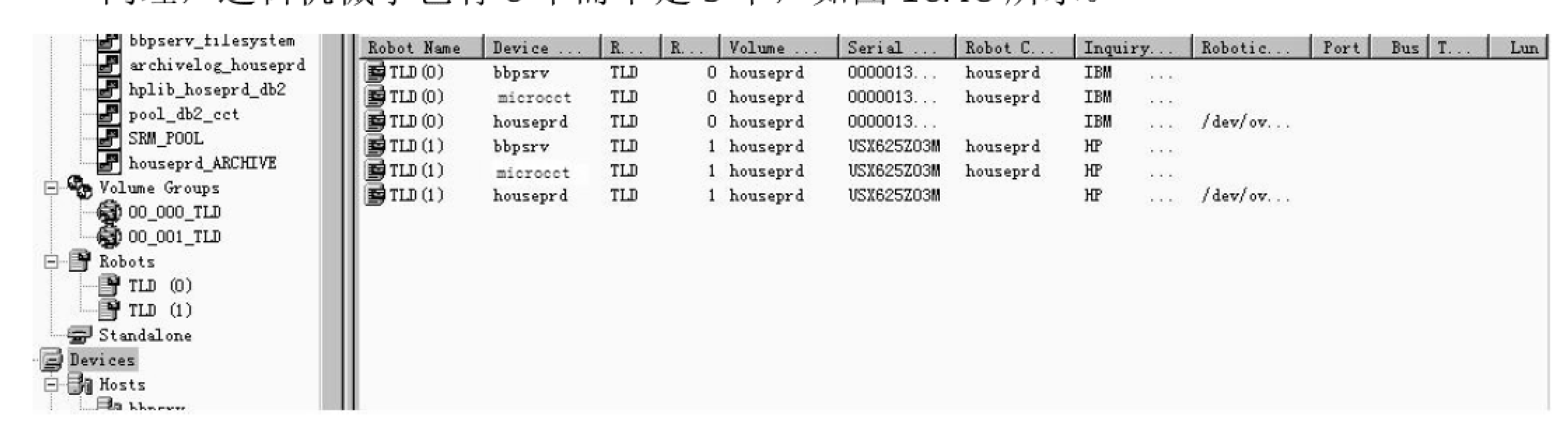


图16.48 逻辑机械手

如果选中 Devices 项下面的某台主机设备，便可在右侧窗口的下半部分显示这台主机掌管的驱动器或者机械手，如图 16.49 所示。

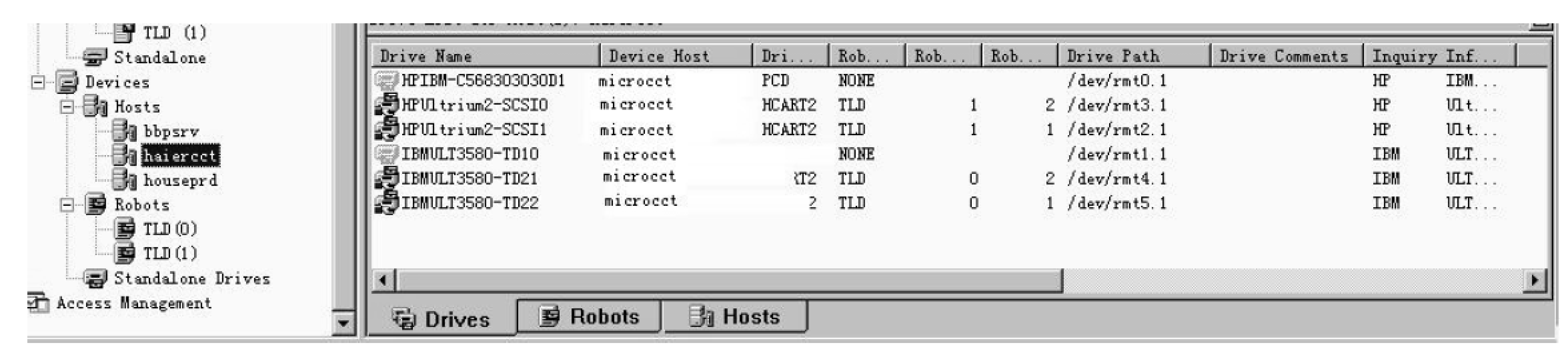


图16.49 每台主机包含的设备

并且在右侧窗口的上半部分，就会突出这台设备所连接的连线，如图 16.50 所示。

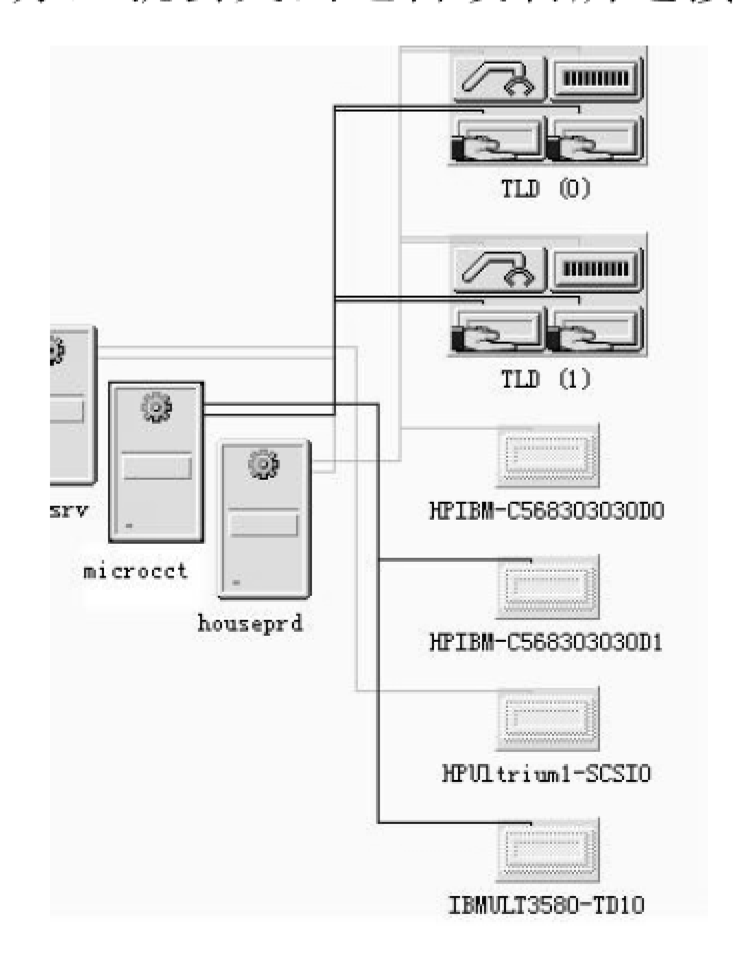


图16.50 拓扑连线

图中黑色连线表示：microcct 主机目前所连接的设备有：两台独立磁带机驱动器、两台磁带库中的共享驱动器。

8. Standalone Drivers(独立驱动器)

如果某台介质服务器上有自己的独立磁带机(一个驱动器，没有机械手)，则 NetBackup 识别到之后，就会在这个项目下显示出来，本例中共有 4 台独立磁带机，如图 16.51 所示。

以上介绍了 NetBackup 配置工具的一些基本组成。下面通过一个实例来说明如何备份 bbpsrv 这台服务器上的 DB2 数据库。

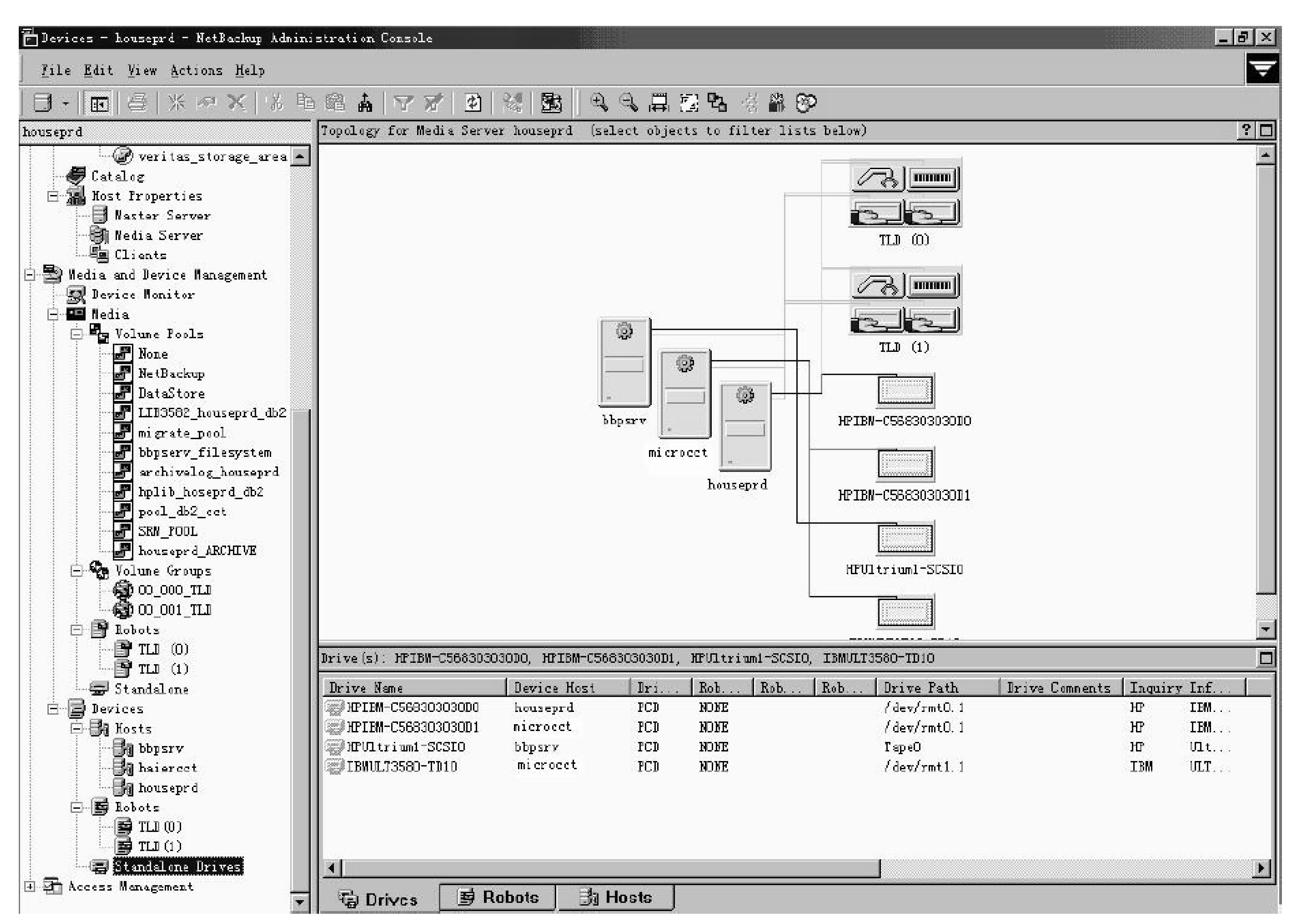


图16.51 独立磁带机列表及拓扑



拓扑图中所有独立磁带机的连接线都用黑色加重了。

16.3.8 配置 DB2 数据库备份

1. 建立备份策略

1） 首先建立一个备份策略，命名为“bbpsrv_db2_bak”，如图 16.52 所示。

2） 单击 New Policy，输入名称之后，显示如图 16.53 所示的对话框。

3） 在 Policy Type 中我们选择“DB2”，使 NetBackup 调用与 DB2 备份相关的模块。在 Policy Volume Pool 下，选中专门为这个备份所创建的卷池“bbpsrv”。

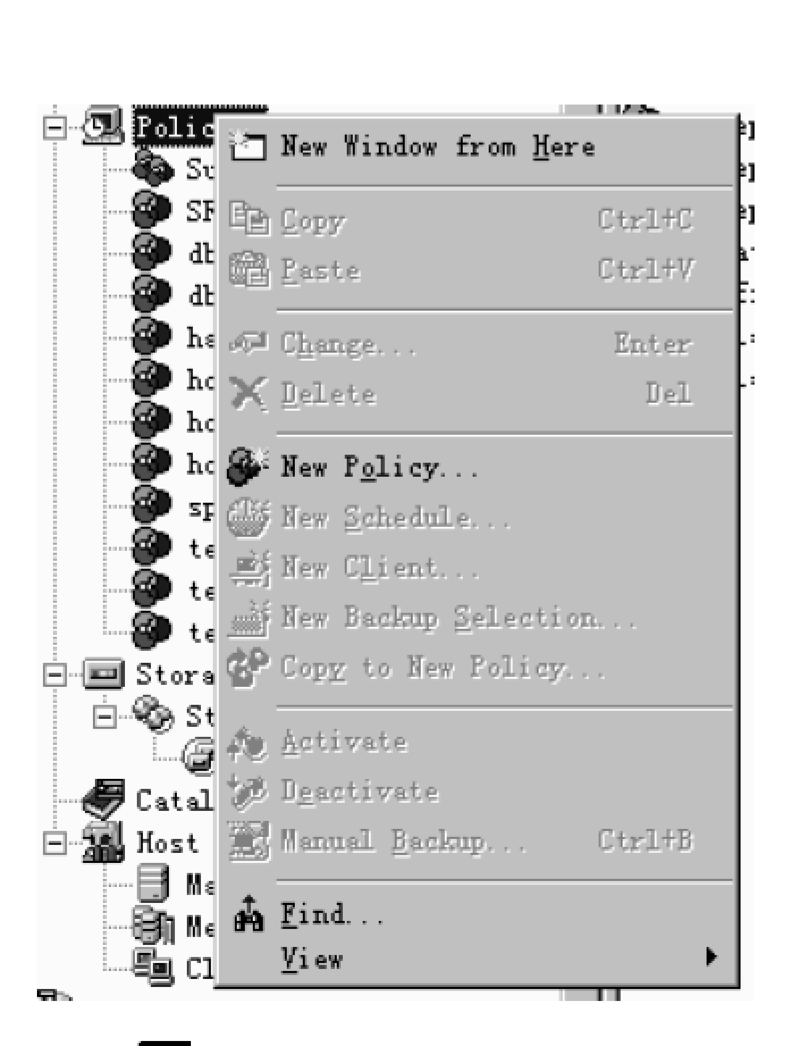


图16.52 创建备份策略

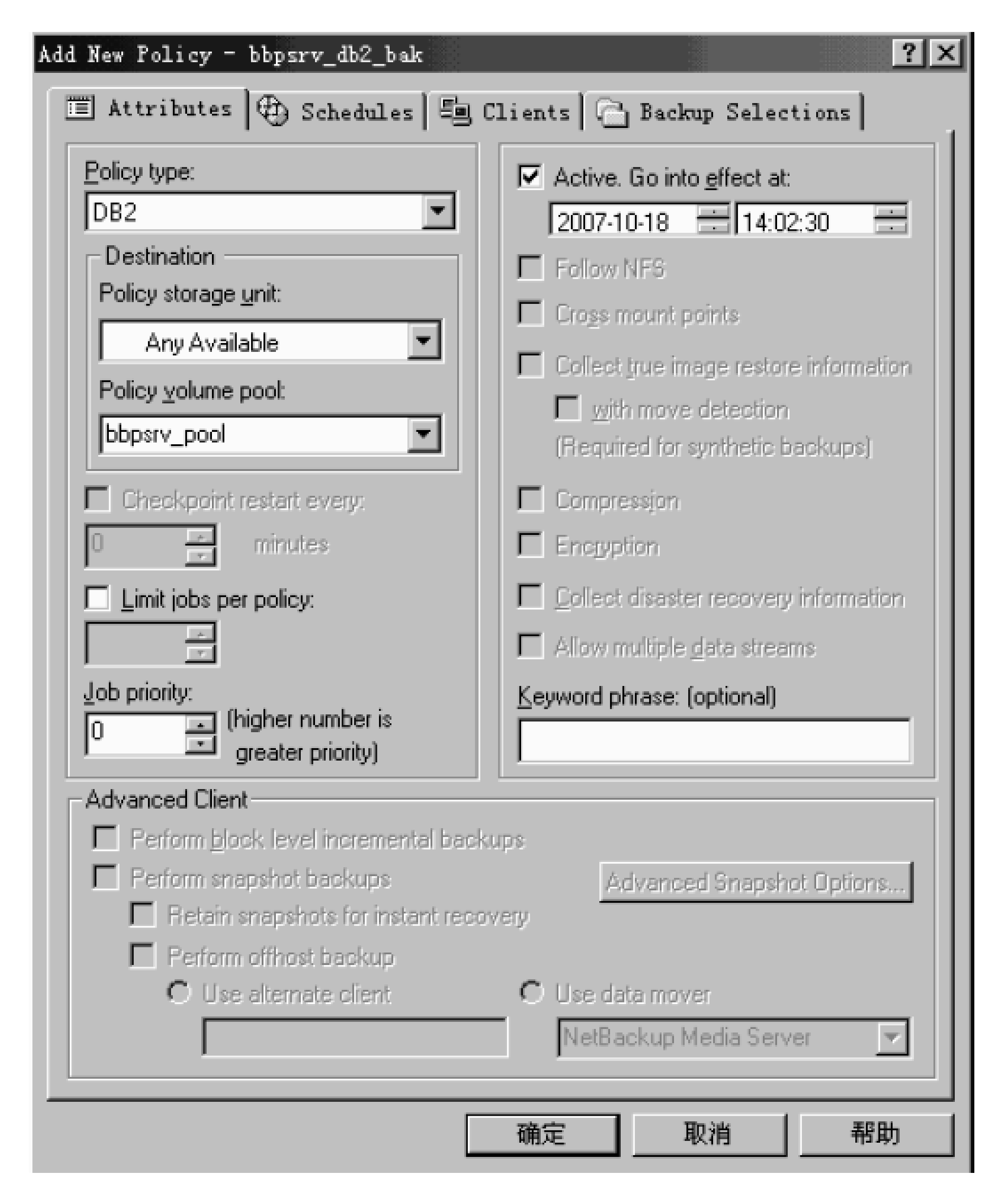


图16.53 新策略窗口

2. 在策略中添加时间表

切换到 Schedules 选项卡，设置什么时间进行数据备份，以及每次备份最多允许花费多长的时间等。

1） 进入 Schedules 选项卡，窗口中已经包含了一个名为：Default-Application-Backup 的 Schedule，这个 Schedule 是备份 DB2 数据库所必须的，因为备份时需要调用的脚本中的 Schedule 名称就是 Default-Application-Backup。我们双击这个 Schedule，弹出如图 16.54 所示的对话框。

其中 Type of backup 为 Application Backup，表明这个 Schedule 是用于由应用程序自主发起的备份。如果没有这个 Schedule，则应用程序就不能调用 Netbackup 提供的接口而把数据发送给 NetBackup，因为策略中没有允许应用程序这么做。

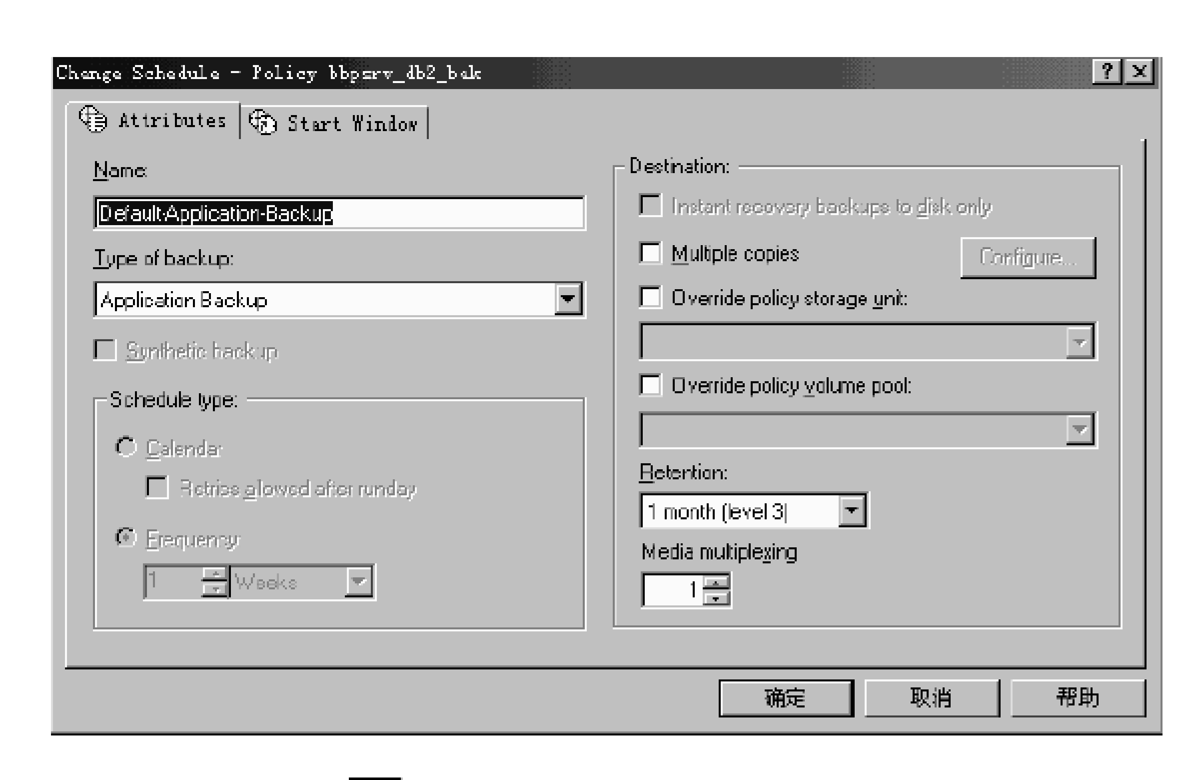


图16.54 编辑策略属性

2】 我们所要实现的，不仅仅是手动从应用程序发起备份，而是让 NetBackup 自动根据设定的时间来备份，所以需要增加一个 Schedule。单击下方的 New 按钮。

3】 将这个 Schedule 命名为“Auto_Full”，Type of backup 选择 Automatic Full Backup，如图 16.55 所示。

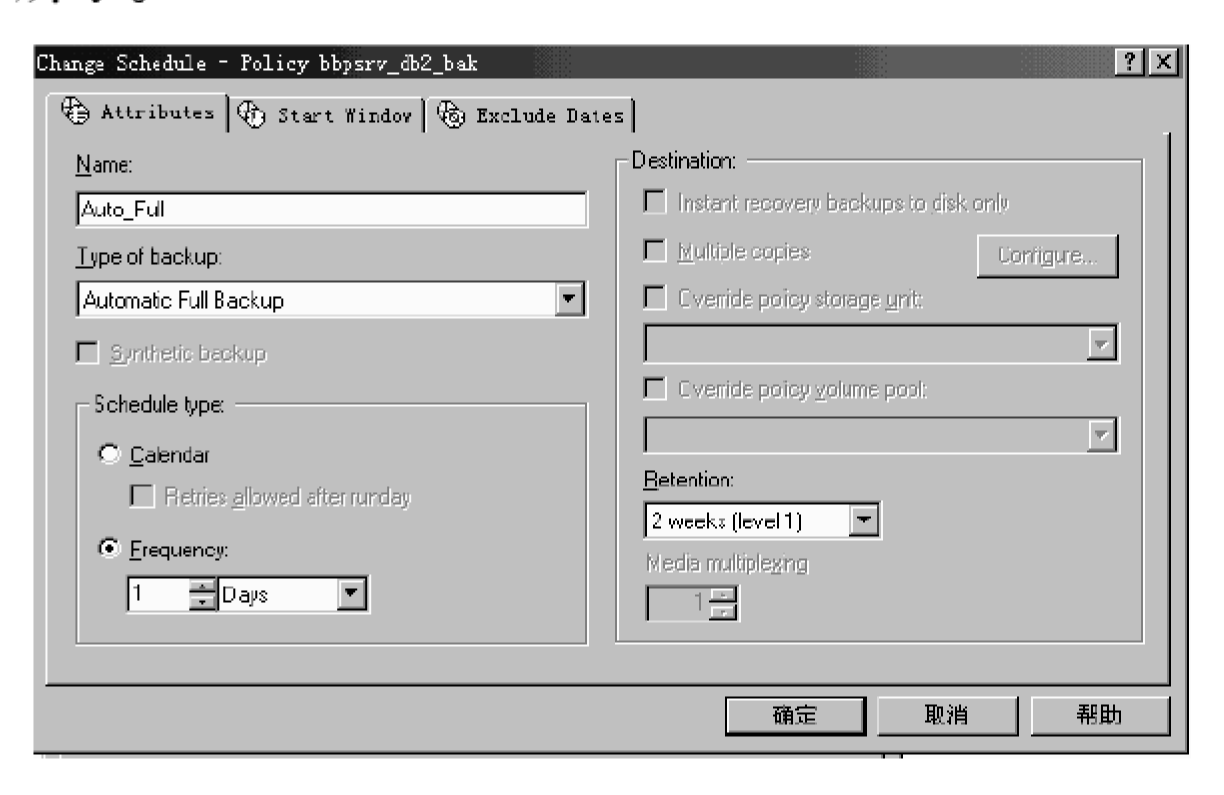


图16.55 给日程起名

4】 Frequency 选择每天备份一次。Retention(保留)选择将备份保留两周，两周后，对应的磁带就可以被抹掉或者用于其他备份。然后切换到 Start Windows 选项卡，如图 16.56 所示。

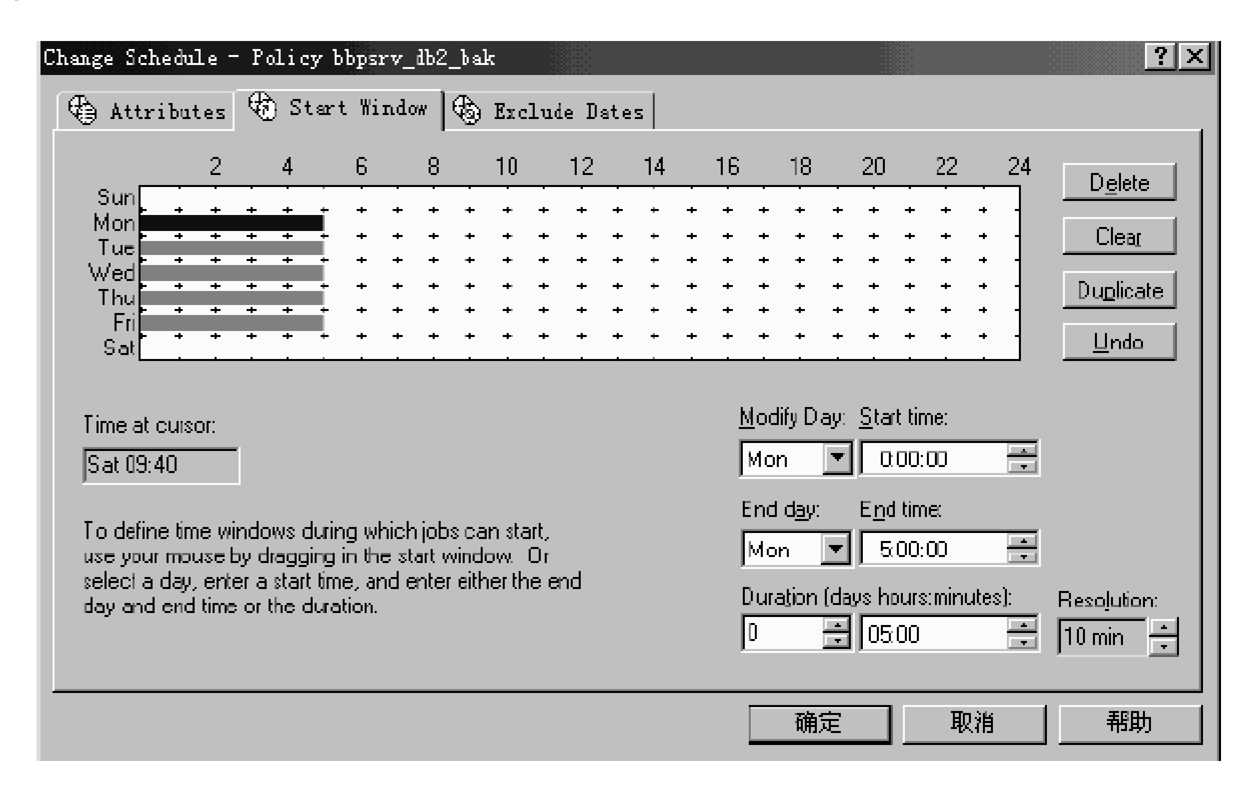


图16.56 选择具体备份窗口时间

5】 这里设置，每天 0 点开始备份，凌晨 5 点结束备份。如果由于某种原因，备份持

续了超过 5 小时，则 NetBackup 会执行完当前备份。如果还有其他备份需要在这 5 个小时中执行，则禁止其执行，直到第二天的 0 点，再接着执行上次未执行的备份，以次类推。

3. 选定需要备份的客户机

1】 接着切换到 Clients 选项卡，如图 16.57 所示。

2】 单击 New 按钮，浏览或者输入要备份的服务器，即 bbpsrv 这台计算机。期间会提示选择这台计算机的操作系统类型，这里选择 Windows 2000。然后切换到 Backup Selections 选项卡。在前三个选项卡中，已经定义了备份类型、备份发生的时间和持续时间、所需备份的服务器，而唯独缺少了最重要的内容，即备份这台服务器上的哪些东西？在 Backup Selections 选项卡来完成这个策略的最后一步：定义备份哪些内容。

4. 选择需要备份的内容或者需要执行的脚本

1】 在 Clients 选项卡中单击 New 按钮，出现如图 16.58 所示的界面。

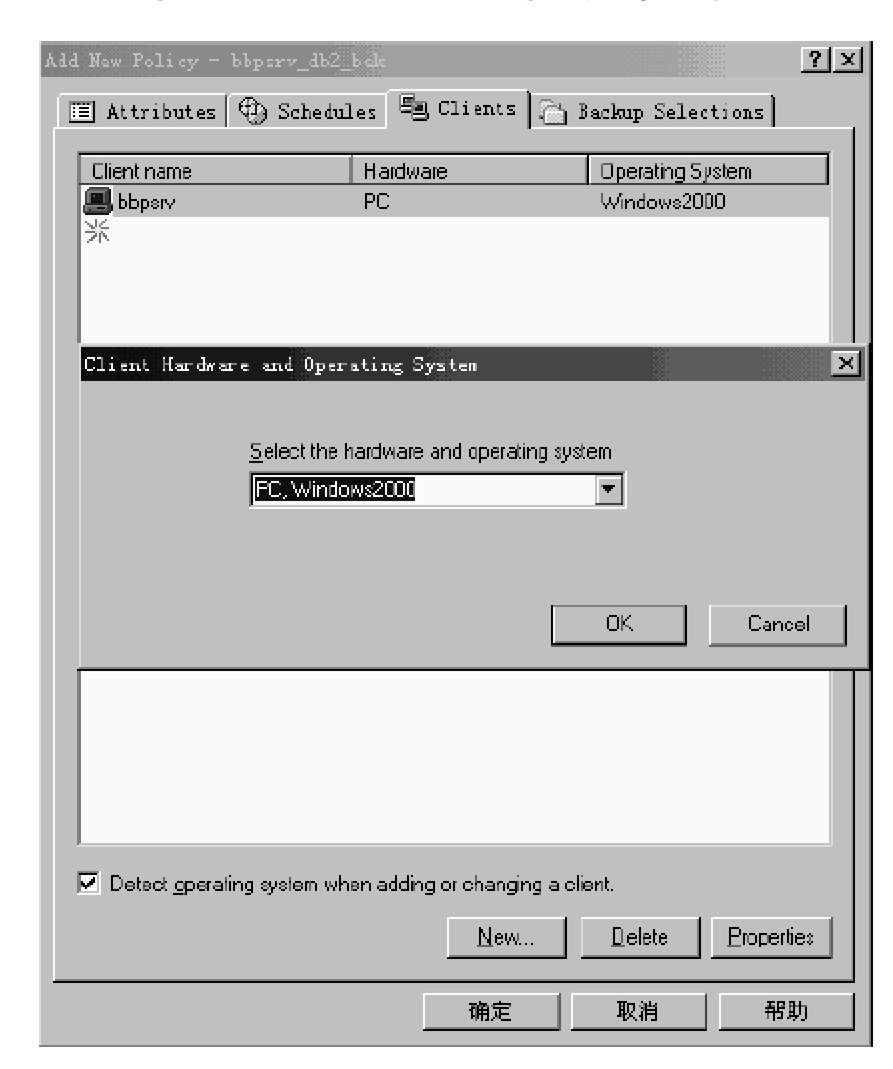


图16.57　设置要备份的客户端



图16.58　选择备份内容或者需要执行的备份脚本

2】 然后单击按钮，来浏览客户机上的文件，如图 16.59 所示。

3】 选中需要备份的目录或者文件之后，单击“确定”按钮。

本例需要备份的是数据库数据，备份数据库如果只备份数据文件，恢复的时候是不够的，况且，如果是 online 备份，则必须用数据库自己提供的工具来备份，才会得到可用的镜像，仅仅把数据文件复制一份，这种备份是不能用作恢复的。所以这个例子中，需要在待备份的计算机上运行 DB2 数据库相关的备份命令来备份数据库。这些命令都存在于一个预先由 NetBackup 编辑好的批处理脚本文件中。

4】 找到这个文件，其路径位于待备份计算机 NetBackup 安装目录下。

```
C:\Program Files\VERITAS\NetBackup\DbExt\DB2\db2_backup_db_online.cmd
```

选中这个文件，单击“确定”按钮，如图 16.60 所示。

可以看到，NetBackup 已经识别出这个脚本，左侧的图标已经变为 。

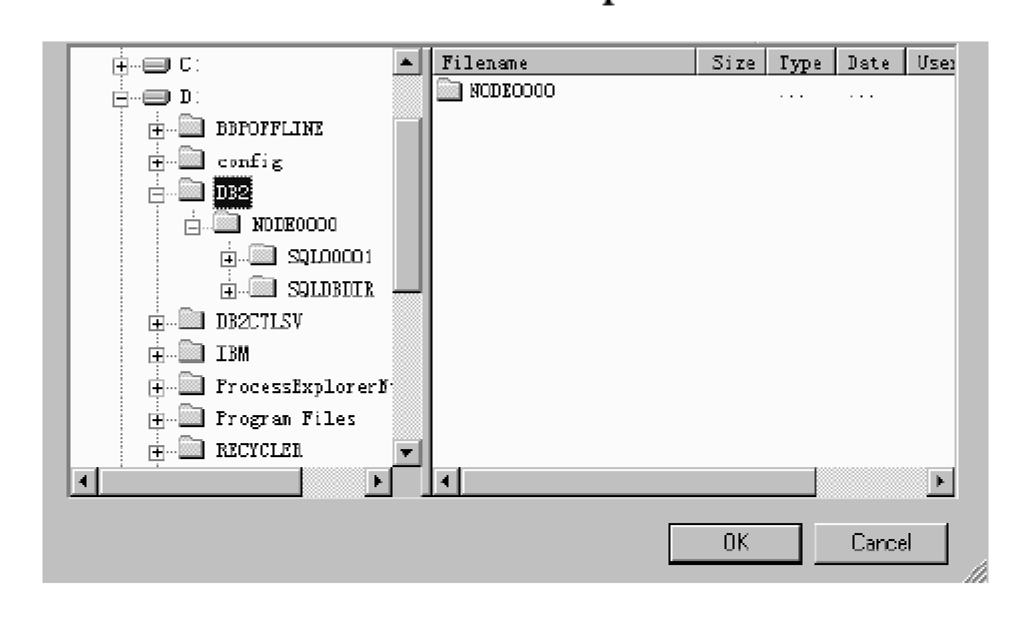


图16.59 选择要备份的文件或者要执行的脚本

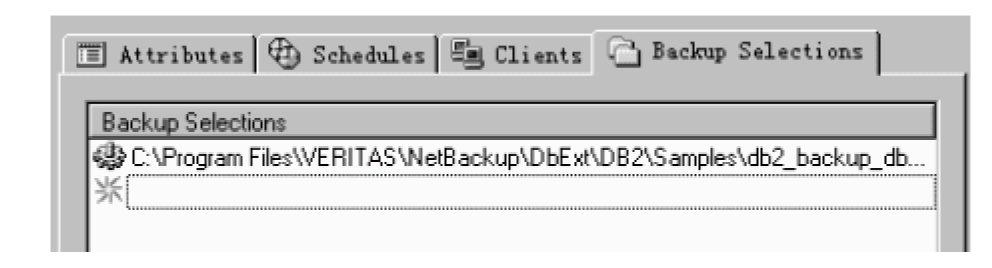


图16.60 选择要执行的脚本

脚本的内容如下：

```
@REM $Revision: 1.2 $

@REM bcpyrght
@REM
************************************************************************
@REM * $VRTScprght: Copyright 1993 - 2003 VERITAS Software Corporation, All
Rights Reserved $ *
@REM
************************************************************************
@REM ecpyrght

@REM
------------------------------------------------------------------------
@REM
@REM This script is provided as an example.  See the instructions below
@REM for making customizations to work within your environment.
@REM
@REM Please copy this script to a safe location before customizing it.
@REM Modifications to the original files will be lost during product updates.
@REM
@REM This script performs an online backup of the database.  An online backup
@REM requires that the database is configured for forward recovery (see the
@REM DB2 USEREXIT and LOGRETAIN settings).  DB2 users can remain connected
@REM while performing an online backup.
@REM
@REM To back up a database or a database partition, the user must have SYSADM,
@REM SYSCTRL, or SYSMAINT authority.
@REM
------------------------------------------------------------------------

@echo off
@setlocal

@REM !!!!! START CUSTOMIZATIONS !!!!!
@REM
@REM The following changes need to be made to make this script work with your
@REM environment:
@REM
@REM
------------------------------------------------------------------------
@REM (1) NetBackup for DB2 shared library:
@REM
------------------------------------------------------------------------
@REM    This is the NetBackup library that backs up and restores DB2 databases
@REM    Set db2_nblib below to the correct NetBackup library path for your
host
```

```
@REM
@REM Example: @set db2_nblib=C:\progra~1\veritas\netbackup\bin\nbdb2.dll
@set db2_nblib=
@echo db2_nblib = %db2_nblib%

@REM
------------------------------------------------------------------------
@REM (2) DB2 home directory (the system catalog node):
@REM
------------------------------------------------------------------------
@REM    This is the DB2 home directory where DB2 is installed
@REM    Set db2_home to DB2 home directory
@REM
@REM Example: @set db2_home=D:\sqllib
@set db2_home=
@echo db2_home = %db2_home%

@REM
------------------------------------------------------------------------
@REM (3) Database to backup:
@REM
------------------------------------------------------------------------
@REM    Set db2_name to the name of the database to backup:
@REM
@REM Example: @set db2_name=SAMPLE
@set db2_name=
@echo db2_name = %db2_name%

@REM
------------------------------------------------------------------------
@REM (4) Multiple Sessions:
@REM
------------------------------------------------------------------------
@REM    Concurrency can improve backup performance of large databases.
@REM    Multiple sessions are used to perform the backup, with each session
@REM    backing up a subset of the database.  The sessions operate
@REM    concurrently, reducing the overall time to backup the database.
@REM    This approach assumes there are adequate resources available, like
@REM    multiple tape devices and/or multiplexing enabled.
@REM
@REM    For more information on configuring NetBackup multiplexing,
@REM    refer to the "Veritas NetBackup System Administrator's Guide".
@REM
@REM    If using multiple sessions change db2_sessions to use multiple
sessions
@REM
@REM Example: @set db2_sessions="OPEN 2 SESSIONS WITH 4 BUFFERS BUFFER 1024"
@set db2_sessions=

@REM !!!!! END CUSTOMIZATIONS !!!!!

@REM
------------------------------------------------------------------------
@REM Exit now if the sample script has not been customized
@REM
------------------------------------------------------------------------
if "%db2_name%" == "" goto custom_err_msg

@REM
------------------------------------------------------------------------
@REM These environmental variables are created by Netbackup (bphdb)
@REM
------------------------------------------------------------------------
```

```
@echo DB2_POLICY = %DB2_POLICY%
@echo DB2_SCHED = %DB2_SCHED%
@echo DB2_CLIENT = %DB2_CLIENT%
@echo DB2_SERVER = %DB2_SERVER%
@echo DB2_USER_INITIATED = %DB2_USER_INITIATED%
@echo DB2_FULL = %DB2_FULL%
@echo DB2_CINC = %DB2_CINC%
@echo DB2_INCR = %DB2_INCR%
@echo DB2_SCHEDULED = %DB2_SCHEDULED%
@echo STATUS_FILE = %STATUS_FILE%

@REM
------------------------------------------------------------------------
@REM Type of Backup:
@REM
------------------------------------------------------------------------
@REM    NetBackup policies for DB2  recognize different
@REM    backup types, i.e. full, cumulative, and differential.
@REM    For more information on NetBackup backup types, please refer to the
@REM    NetBackup for DB2 System Administrator's Guide.
@REM
@REM    Use NetBackup variables to set DB2 full or incremental options
@REM

@set db2_action=
if "%DB2_FULL%" == "1" @set db2_action=ONLINE
if "%DB2_CINC%" == "1" @set db2_action=ONLINE INCREMENTAL
if "%DB2_INCR%" == "1" @set db2_action=ONLINE INCREMENTAL DELTA
@echo db2_action = %db2_action%

@REM
------------------------------------------------------------------------
@REM Actual command that will be used to execute a backup
@REM Note: the parameters /c /w /i and db2 should be used with db2cmd.exe
@REM Without them, NetBackup job monitor may not function properly.
@REM
------------------------------------------------------------------------

@set CMD_FILE=%temp%\cmd_file
@echo CMD_FILE = %CMD_FILE%

@set CMD_LINE=%db2_home%\bin\db2cmd.exe /c /w /i db2 -f %CMD_FILE%
@echo CMD_LINE = %CMD_LINE%

@echo BACKUP DATABASE %db2_name% %db2_action% LOAD %db2_nblib%
%db2_sessions%
@echo BACKUP DATABASE %db2_name% %db2_action% LOAD %db2_nblib%
%db2_sessions% > %CMD_FILE%

@REM
------------------------------------------------------------------------
@REM Execute the command
@REM
------------------------------------------------------------------------

@echo Executing CMD=%CMD_LINE%

%CMD_LINE%

@REM Successful Backup
if errorlevel 1 goto errormsg
echo BACKUP SUCCESSFUL
if "%STATUS_FILE%" == "" goto end
if exist "%STATUS_FILE%" echo 0 > "%STATUS_FILE%"
```

```
goto end

:custom_err_msg
echo This script must be customized for proper operation in your environment.

@REM Backup command unsuccessful
:errormsg
echo Execution of BACKUP command FAILED - exiting
if "%STATUS_FILE%" == "" goto end
if exist "%STATUS_FILE%" echo 1 > "%STATUS_FILE%"

:end

@endlocal
```

经过这样的配置之后，在每天的 0 点，Master Server 便会发送指令给 bbpsrv 上的 NetBackup 客户端，让它执行这个脚本，此脚本中的命令，会告诉 DB2 数据库备份数据库并且调用一个 DLL 链接库文件，将数据通过 SAN 网络发送到相应卷池所在的磁带库上，从而被写入磁带。

5. 监控备份执行状况

可以通过下面的方法监控备份执行的状况，如图 16.61 所示。

右侧窗口的下半部分显示了备份运行的状况，如果成功备份，则显示一个蓝色小人成功举起的图标；如果备份正在运行，则显示一个绿色小人正在奔跑的图标；如果备份任务正在等待，则显示三个绿色小人排队的图标；如果任务失败，则显示圆形红色差号。

对于其他服务器，可以用同样的步骤来备份，这里就不再赘述了。

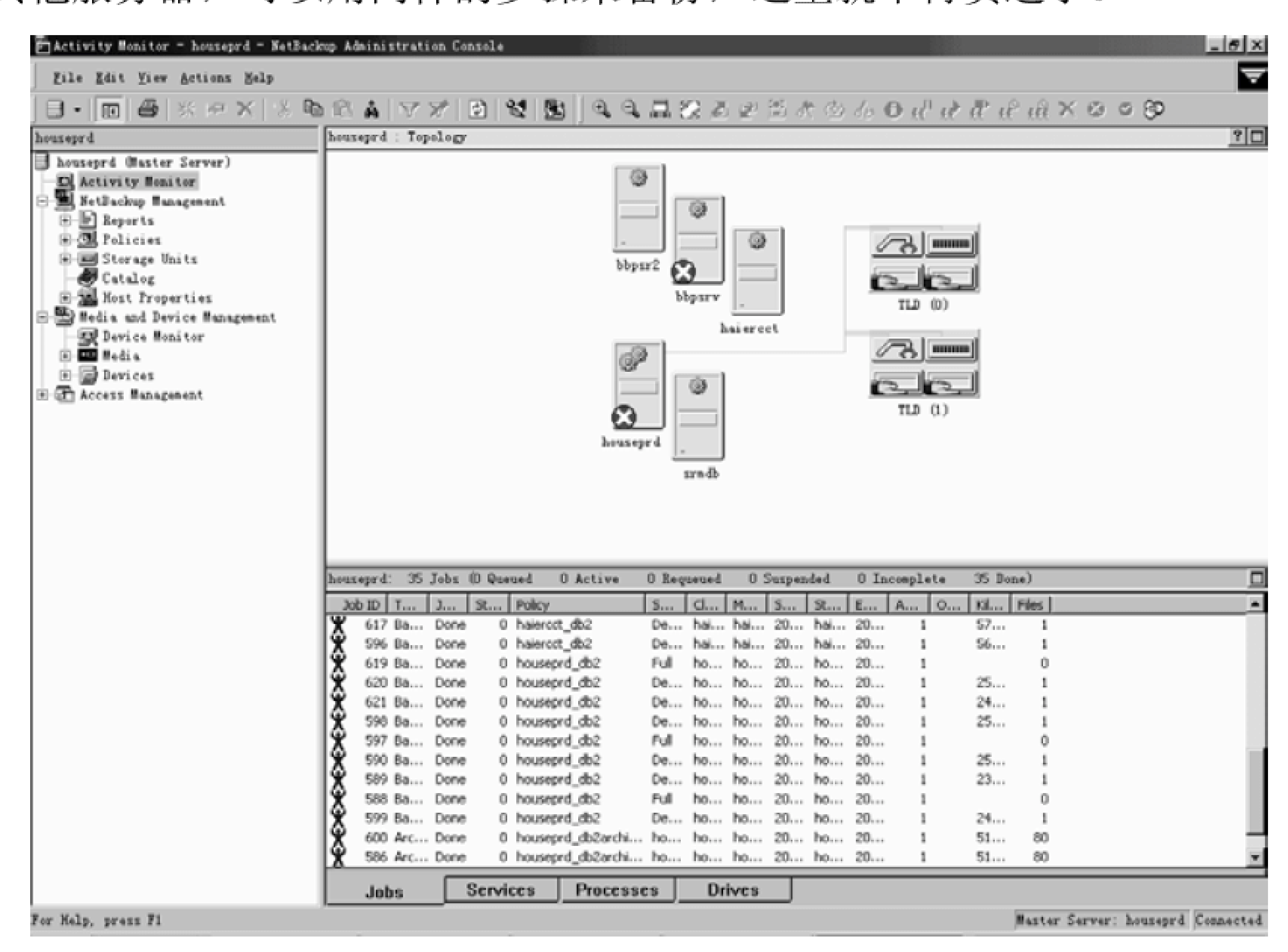
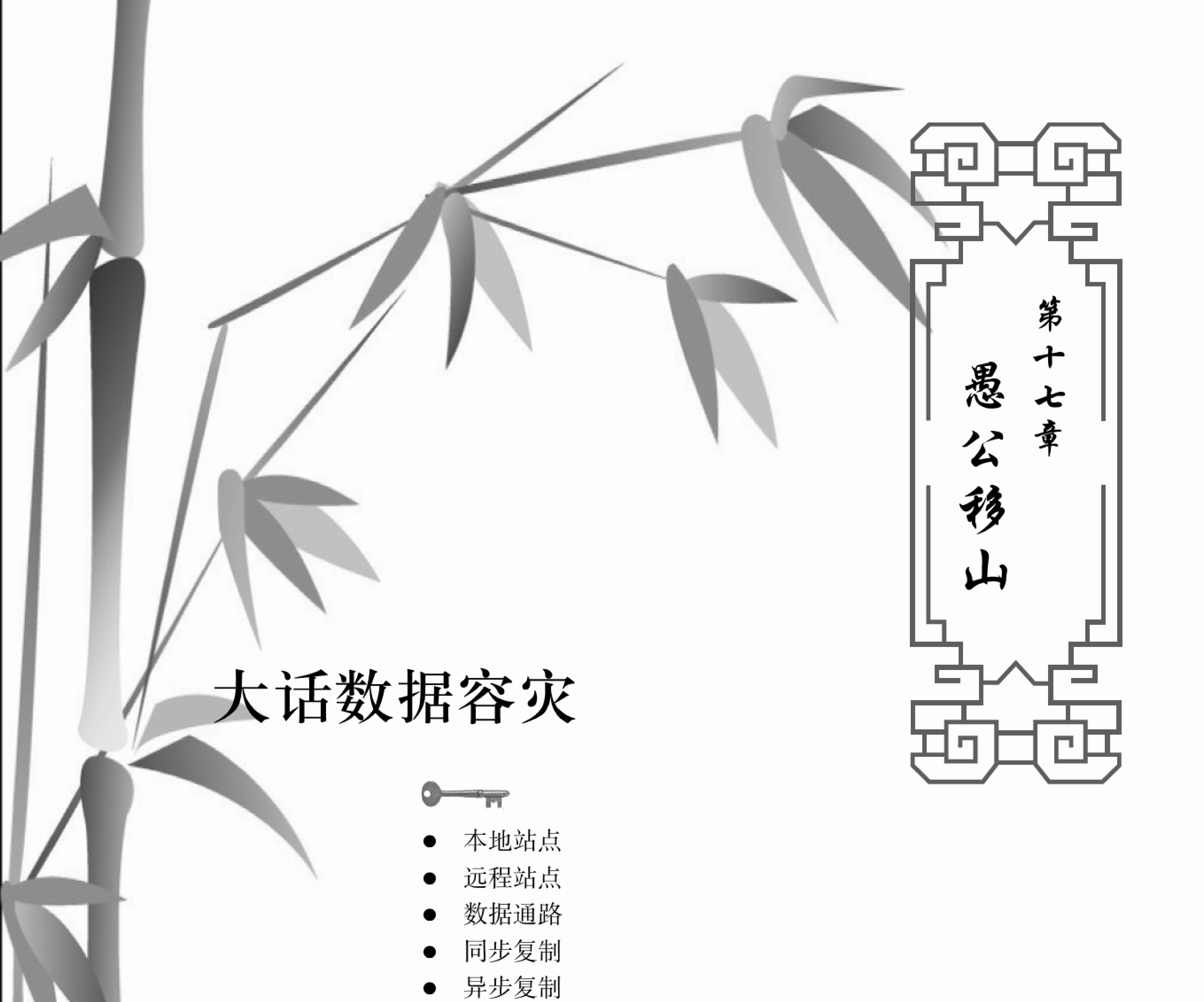


图16.61 监控备份状态



大话数据容灾

- 本地站点
- 远程站点
- 数据通路
- 同步复制
- 异步复制
- 基于主机的数据复制
- 基于存储的数据复制

数据备份系统只能保证数据被安全的复制了一份，但是一旦生产系统发生故障，比如服务器磁盘损坏致使数据无法读写、主板损坏造成直接无法开机或者机房火灾等意外事件，我们必须将备份的数据尽快地恢复到生产系统中继续生产。这个动作，就叫做容灾。

提示：容灾可以在出现故障后手动完成，也可以靠程序自动完成，

17.1 容灾概述

有些事件中，很多公司就是因为没有远程容灾系统，导致数据全部毁于一旦，客户数据丢失、公司倒闭，受损失的不仅是公司，还有客户。如果要充分保障系统和数据的安全，只是在本地将数据进行备份还远远不够，还必须在远程地点建立另外一个系统，并包含当前生产系统的全部数据备份。这样在本地系统发生故障的时候，远程备份系统可以启动，继续生产。

要实现这样一个系统，首先，要保证主生产系统的所有数据实时的传输到远程备份系统。

其次，主系统发生故障之后，必须将应用程序也切换到远程备份系统上继续运行。应用程序是一个企业生产流程的代码化表示，只有应用程序正在运行，这个企业才处于生产过程中，而应用程序的成功运行，又必须依赖于底层数据。

俗话说，巧妇难为无米之炊。我们的应用程序，比如 Exchange 邮件转发系统、SAP 企业 ERP 系统、Lotus Notes 办公自动化系统等，这些就好比巧妇，而保存在磁盘上的数据，比如用户的邮件、ERP 系统的数据库文件、办公自动化系统自身数据文件等，就好比大米，巧妇用她高超的厨艺，将大米做成熟饭，供消费者购买。

这就是一个企业生产的基本雏形，企业(巧妇)用应用程序(高超的厨艺)来处理各种数据(大米)，最终生成新的数据(米饭)，供消费者购买。而巧妇所利用的锅碗瓢盆、水、电、煤气等也是必不可少，比如服务器、硬盘、网络通信设施、电源等这些 IT 系统必要组件。下面来对比一下厨房和 IT 系统机房，如表 17.1 所示。

表 17.1　厨房与机房

	厨　房	机　房
生产工具	锅碗瓢盆、炉灶、铲勺	服务器、硬盘、网络通信设施、电源等
生产资料	大米、面粉、蔬菜，油盐酱醋	录入的原始数据
生产者	厨师	各种应用程序逻辑
产品	美味菜肴	客户需要的信息

生产者厨师，用生产工具来加工生产资料，获得产品。同样，各种应用程序，运行在服务器上，将各种原始数据加工修改，产生客户需要的信息。二者在本质上是相同的。

基于这个生产模型，我们把一个企业 IT 生产系统划分为四个组件：生产资料、生产工具、生产者、产品。要实现整个 IT 系统的容灾，那么必然要实现上述所有四个组件的容灾。然而，IT 系统的产品和原始数据往往都存放在同一位置，比如同一个卷，同一台盘阵等。本章不描述产品的容灾，因为其与生产资料的容灾本质是相同的。

思考

厨房的容灾。大家不要笑，厨房容灾，这个名词是不是太荒唐了。现在貌似是的，但是战争年代，厨房容灾将是必备的。人是铁饭是钢，关键时刻看厨房。我想厨房容灾这个话题，广大读者应该比 IT 工作者的办法更多了。比如，在另外一个隐蔽地点建立一个厨房，柴米油盐酱醋茶，锅碗瓢盆炉灶勺都储存到这个厨房中作为储备粮和储备工具，一旦当前厨房被敌人摧毁，立即启用备用厨房，厨师全部转移到备用厨房继续做饭。这没啥难度。的确，谁都可以想出这样的办法。那么我们看看能否用这个思想来建立 IT 系统的容灾。

生产工具的容灾

像厨房容灾一样，在另一个地点建立一个 IT 机房，服务器、网络设施、磁盘阵列设施等一应俱全，当然，出于成本因素，备用地点的设备不一定非要与主系统中的生产工具规格和性能完全相同，在性能和容量上的要求可以适当降低，但至少要满足生产需求。

下面将主要讲解生产资料的容灾和生产者的容灾。

17.2 生产资料容灾——原始数据的容灾

IT 系统的生产资料，即各种原始录入数据。它和实物化的生产资料比如大米，有很大不同。

第一，IT 数据是可以任意复制，并可以复制多份的数据。

第二，IT 系统数据是不断变化的，在生产的同时，原始数据将会不断的变化，甚至产品数据会覆盖原始数据。

基于 IT 数据的这二个特点，在 IT 系统的生产资料容灾方面，需要注意以下两点。

第一，不可能像储存大米一样，把某时刻的原始数据复制到备用系统中就不管了，因为这份数据是不断在变化中的，我们需要把变化，实时的同步到备用系统中，只要主系统数据变化了，备用系统的数据也要跟着变化。

第二，数据必须至少保留额外的一份。因为大米没有了，可以再购买，每次购买的大米也可以一样的。但是如果数据没有了，就不可再生，这个企业就要面临倒闭。所以在实现容灾的同时，还必须做好数据备份工作，将数据备份到磁带或者其他备份目标中保存。基于 IT 系统数据是不断变化的，所以需要尽量保存这份数据的最后状态，比如，一天备份一次。如果数据量很小，甚至可以一天备份多次，这样可以充分保证备份的数据与当前的数据相差最少。

有了以上两点的保证，就可以像厨房容灾一样，来设计 IT 系统的生产资料容灾了，以下是一个设计好的例子。

1】 用网络来连接本地系统和备用系统，先将本地系统某时刻的数据，实时传送到备用系统。

2】 传输结束后，再将从这个时刻之后的所有变化的数据，同步到备用系统。

3】 此后，只要本地系统有数据变化，则立即将变化的数据传送到备用系统，使备用系统数据发生相同的变化。

在这个基础上，还需要在本地系统中对数据做额外备份，即备份到离线(如磁带等)介质

上，做到时刻保留一份额外的数据。有条件的话，还可以将备份好的磁带运送到备份站点去，这样就充分保证了主站点一旦发生火灾等全损型故障后的数据冗余度。

生产资料的容灾，是容灾系统中最重要的一个组件，因为没有了生产资料，纵使有生产工具、生产者，也无法进行生产，而如果没有了生产工具和生产者，比如服务器、应用软件等，则可以很容易购买到。

将生产数据通过网络实时传送到备用系统，要实现这个目的，要怎么来设计呢？我们来看个拓扑图，如图 17.1 所示。

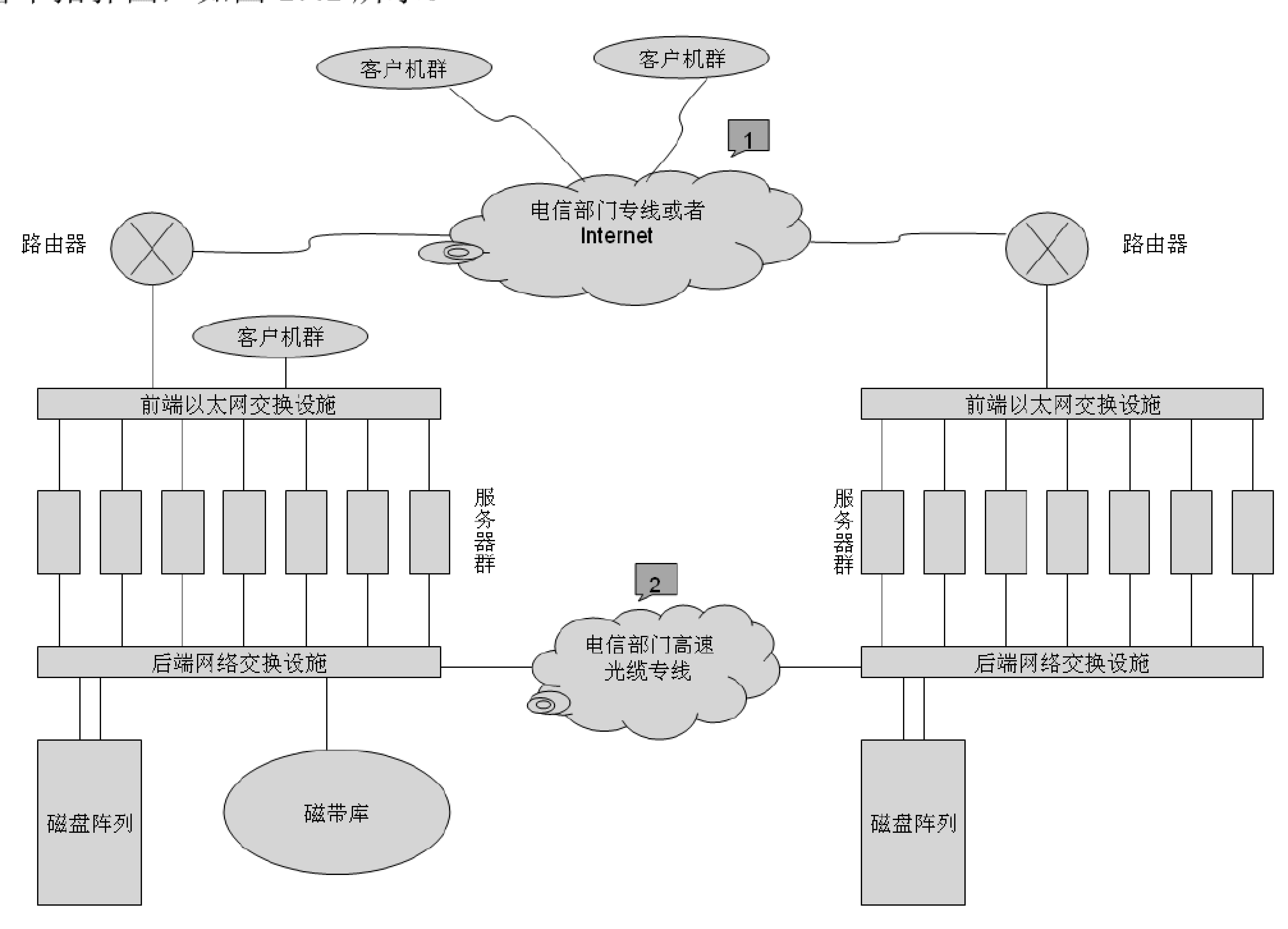


图 17.1　两种数据通路的位置

左边的主站点和右边的备份站点存在相同的生产工具。既然要使主站点和备用站点之间必须通过网络来接连，所以只要知道究竟在哪个网络设施上进行连接。系统中有两个网络通信设施，一个是前端以太网络，另一个是后端 SAN 网络(也有可能是以太网络，即服务器通过 ISCSI 协议连接到后端磁盘阵列)，我们究竟是连接二者的前端网络还是连接后端网络呢(图中标注的两条路径)？答案是，两者都可以。

17.2.1　通过主机软件实现前端专用网络或者前端公用网络同步

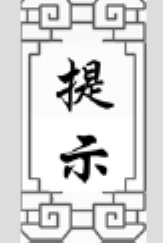
这种方式利用的就是图 17.1 中的标注 1 所示的路径。

1】 主站点和备站点的前端以太网络，均通过路由器连接至电信部门的专线或者 Internet 网络上。

2】 主站点上变化的数据，经过前端以太网交换机，然后通过路由器，传送给电信部门的网络交换路由机组中，经过层层交换或路由，传输到备站点的路由器。

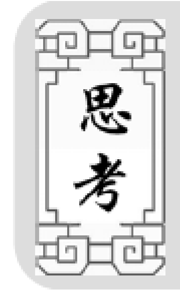
3】 然后经过交换机传送到备站点的相应服务器上。

4】 服务器收到数据后，写入后端的磁盘阵列上。



如果为了连接而接入 Internet 网络，则最好做成 VPN 模式，隧道的基础上，如果追求数据安全性，则需要配置加密模式的 VPN。如果对数据同步的实时性要求不高，而数据量又很大的情况下，主站点和备站点都接入 100Mb/s 的 Internet 网络，反而会得到很大的实惠，特别是备站点和主站点在相同城市的情况下，这样主站点数据路由到电信部门的设备之后，会经过很少的设备就会到达备站点，而且能保证很大的实际带宽。如果是利用窄带专线接入专网，虽然可以保证这条链路带宽独享，但是毕竟带宽低。所以专线可以保证数据同步的实时性，但是不适合大数据量的传输。

因为这种方式同步数据，需要经过前端网络，所以实现这个功能，还需要在距离前端网络最近的主机设备上实现，也就是服务器群上来实现。



为何不能直接在网络交换设备上或者路由器上来实现呢？

第一，网络设备一般是没有灵活的程序载入运行能力的，网络设备上运行的程序都是预先固化到芯片中的程序，一般不可修改，更不用说再加入另外的程序来执行了。

第二，数据存在于服务器，或者连接服务器的磁盘阵列上，网络设备若想从服务器上提取数据，则必须通过调用操作系统提供的相关程序接口。而跨网络传输数据的现成接口，只有网络文件系统，所以还需要将所有数据卷都共享成为 NAS 模式，但这样过于繁冗，所以我们必须要在每台有数据备份的服务器上安装一种软件来实现数据的同步，将这种软件安装在生产者，也就是应用服务器上，然后在服务器上提取生产资料和产品。

不管生产资料和产品存在于服务器本地磁盘，还是存在于后端的磁盘阵列上，对于服务器操作系统来说，都是一个个的目录。在第 15 章的最后曾经描述过操作系统的目录虚拟化，不管底层用的是什么设备来存储数据，亦不管数据存放在网络上还是本地磁盘上，最终操作系统都将这些位置虚拟成目录，比如 Windows 的 C 盘、D 盘，或者 UNIX 下的/mnt、/mountpoint 等。

通过这种软件可以直接监视这些目录中数据的变化，只要有变化的数据，就提取出来通过网络传送到远端服务器上，远端服务器同样需要安装这种软件的接收端模块来接收数据，并写入远端备份站点上相应服务器的相同目录。

这种方式利用了前端网络进行数据同步。一般前端网络相对后端网络速度来说是比较低的网络，而且是客户用来访问服务器的必经之路，所以它的资源是比较宝贵的。另外，前端网络一般都是以太网，相对廉价，而且容易整合到企业大网络中，从而接入电信部门的网络，所以适合基于 TCP/IP 协议的远距离传输，比如大于 100 公里的范围，甚至跨国界的 Internet 范围内传输。

下面来看一下这种方式下的数据流经路径。

本地磁盘阵列(或者本机磁盘)－本地后端网络交换设施－本地服务器内存－本地前端

网络－电信交换机组－远端前端网络－远端服务器内存－远端后端网络交换设施－远端磁盘阵列(或者远端本机磁盘)。

如果数据源在本机磁盘，则会跳过本地后端网络交换设施，直接到服务器内存。如果数据是直接在内存中生成的，则需要写入本地一份，同时发送给远端一份保存。其中，在“本地磁盘阵列－本地后端网络交换设施－本地服务器内存”这段路径上，数据是通过 FCP 协议(SCSI over FC 协议)进行打包传送的；在“本地前端网络－电信交换机组－远端前端网络”这段路径上，数据是通过 TCP/IP 协议传送的。FCP 协议运行在后端高速网络的保障之上，而 TCP/IP 协议运行在使用前端低速网络的设备上，保障数据传输，二者各得其所，充分发挥着各自的作用。

图 17.2 所示的路径即是数据流动的路径。服务器上的涡轮泵表示数据同步软件，它从本地提取数据，并将数据源源不断的发送给远端，远端服务器上的涡轮泵将收到的数据源源不断的写入存储设备。

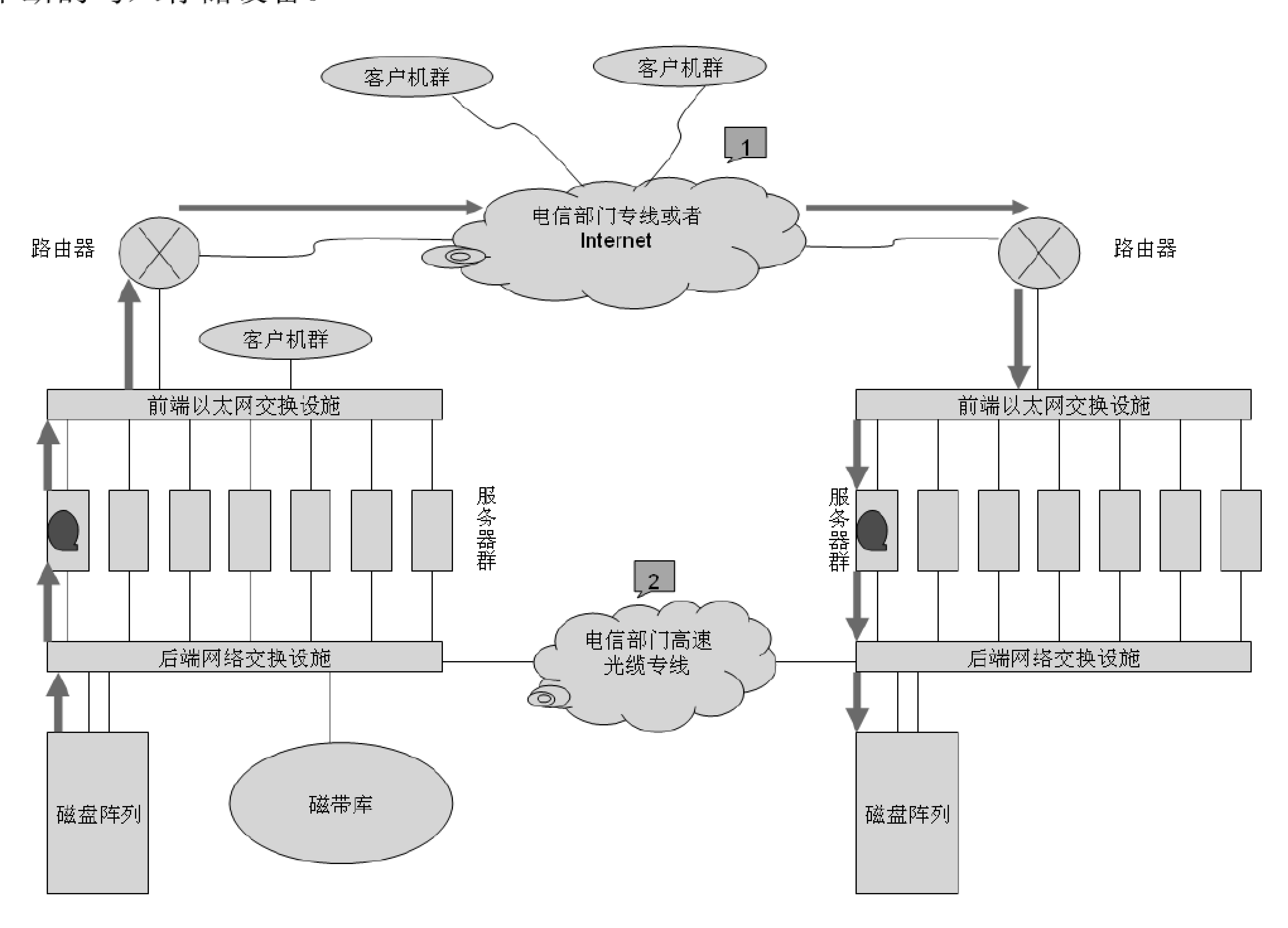


图17.2　经前端网络备份数据

这种方式的数据同步，一般都是文件级同步，即同步软件只检测文件这一层的数据变化，或者每当主服务器针对某个文件做写入时，写入数据会同时发送给备用服务器。而对底层卷的数据块的变化，不做同步，除非数据块的变化造成了对应的文件的逻辑数据变化。

下面介绍两个使用这种方式来同步数据的案例。

Veritas Volume Replicator 软件介绍

Veritas Volume Replicator(简称 VVR)，是 Veritas 公司容灾套件 Storage Foundation 系

列软件中的一个模块，它的作用非常专一，就是将本地某个卷上的数据变化，通过前端 IP 网络复制到远端对应的卷上，而且保证数据变化发生的顺序不被打乱，完全按照本地的 IO 发生顺序在异地按照相同的顺序重现这个 IO。

VVR 支持耗费带宽调整功能，控制对网络带宽的使用，在业务繁忙时可以降低发送速度来减少对网络带宽的耗费，并且可以针对不同的同步流设定各自的带宽。支持异步和同步复制(在后面会介绍)。在网络发生故障的时候，可以自动将复制模式从同步切换到异步，以减少对主机业务的影响。一旦复制断开，VVR 可以记录主站点自从断开之后的数据变化，待连接重新建立之后，立即复制这些变化的数据，而不需要对两边数据进行重新比对或者全部重新复制。



类似的软件还有很多，比如 Double Take，Legato，国产同步软件 InfoCore Replicator 等，他们的构思都是一样的，只不过实现方式和效果上有所不同。

17.2.2 案例：DB2 数据的 HADR 组件容灾

这个案例是笔者实施的一个 DB2 数据库容灾案例，它利用了运行在主机和备份机上的一个数据库软件模块(即 HADR)，来实现两端的数据同步，主机和备份机之间使用基于以太网的 TCP/IP 协议连接。这个案例同步的数据不是卷上的原始数据，而是一种对数据操作的描述，即数据库日志，比如："在 D 盘创建一个表空间数据文件，名称 testspace，大小 500MB"。

这就是一条日志，主机只需要把这句话告诉备份机即可，而不需要传输 500MB 的数据。备份机收到日志后，便会在备份机的磁盘上重做(replay)这些操作，达到与直接同步卷上数据殊途同归的效果。

1. HADR

全称为 High Availability Disaster Recovery，它是 DB2 数据库级别的高可用性数据复制机制，最初被应用于 Informix 数据库系统中，称为 High Availability Data Replication(HDR)，是 Share-Nothing 方式容灾的典型代表。



在 IBM 收购 Informix 之后，这项技术就应用到了新的 DB2 发行版中。

一个 HADR 环境需要两台数据库服务器：主数据库服务器(primary)和备用数据库服务器(standby)。

- 当主数据库中发生事务操作时，会同时将日志文件通过 TCP/IP 协议传送到备用数据库服务器，然后备用数据库对接受到的日志文件进行重放(Replay)，从而保持与主数据库的一致性。
- 当主数据库发生故障时，备用数据库服务器可以接管主数据库服务器的事务处理。此时，备用数据库服务器作为新的主数据库服务器进行数据库的读写操作，而客户端应用程序的数据库连接可以通过自动客户端重新路由(Automatic Client

Reroute)机制转移到新的主服务器。

- 当原来的主数据库服务器被修复后，又可以作为新的备用数据库服务器加入HADR。通过这种机制，DB2 UDB 实现了数据库的故障恢复和高可用性，最大限度的避免了数据丢失。图 17.3 为 DB2 HADR 的工作原理图。

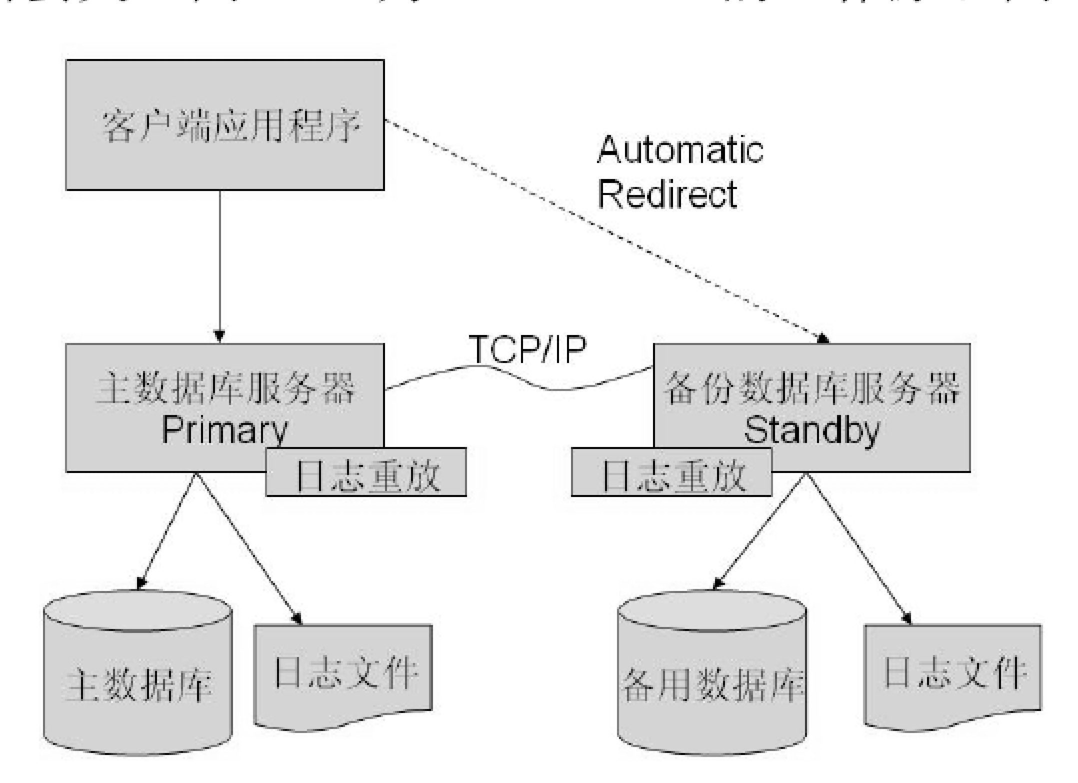


图17.3 DB2 HADR 工作原理图



处于备用角色的数据库不能被访问。

HADR 有三种同步方式。

- SYNC(同步)

  此方式可以尽可能地避免事务丢失，但在三种方式中，使用此方式会导致事务响应时间最长。在此方式中，仅当日志已写入主数据库上的日志文件，而且主数据库已接收到来自备用数据库的应答，确定日志也已写入备用数据库上的日志文件时，方才认为日志写入是成功的。保证日志数据同时存储在这两处。

  如果备用数据库在重放日志记录之前崩溃，则它下次启动时，可从其本地日志文件中检索和重放这些记录。如果主数据库发生故障，故障转移至备用数据库，可以保证任何已在主数据库上落实的事务，也在备用数据库上落实。故障转移操作之后，当客户机重新与新的主数据库连接时，可能会有在新主数据库上已落实的事务，对于原始主数据库却从未报告为已落实。当主数据库在处理来自备用数据库的应答消息之前出现故障时，即会出现此种情况。客户机应用程序应考虑查询数据库以确定是否存在此类事务。

  如果主数据库失去与备用数据库的连接，则不再认为这些数据库处于对等状态，而且将不阻止事务等待来自备用数据库的应答。如果在数据库断开连接时执行故障转移操作，则不保证所有已在主数据库上落实的事务将出现在备用数据库上。

  当数据库处于对等状态时，如果主数据库发生故障，则可以在故障转移操作之后，作为备用数据库重新加入 HADR 对。因为在主数据库接收到来自备用数据库的应答，确认日志已写入备用数据库上的日志文件之前，不认为事务已落实，所以主数据库上的日志顺序将与备用数据库上的日志顺序相同。原始主数据库(现在是备

用数据库)只需要通过重放自从故障转移操作以来，在新的主数据库上生成的新日志记录来进行同步更新。

如果主数据库发生故障时并未处于对等状态，则其日志顺序可能与备用数据库上的日志顺序不同。如果必须执行故障转移操作，主数据库和备用数据库上的日志顺序可能不同，因为在故障转移之后，备用数据库启动自己的日志顺序。因为无法撤销某些操作(比如，删除表)，所以不可能将主数据库回复到创建新的日志顺序的时间点。

如果日志顺序不同，当在原始主数据库上发出指定了 AS STANDBY 选项的 START HADR 命令时，将返回错误消息。如果原始主数据库成功地重新加入 HADR 对，则可以通过发出未指定 BY FORCE 选项的 TAKEOVER HADR 命令来完成数据库的故障恢复。如果原始主数据库无法重新加入 HADR 对，则可以通过复原新的主数据库的备份映像来将此数据库重新初始化为备用数据库。

- NEARSYNC(接近同步)

  此方式具有比同步方式更短的事务响应时间，但针对事务丢失提供的保护也很少。在此方式中，仅当日志记录已写入主数据库上的日志文件，而且主数据库已接收到来自备用系统的应答，确定日志也已写入备用系统上的主存储器时，才认为日志写入是成功的。仅当两处同时发生故障，并且目标位置未将接收到的所有日志数据转移至非易失性存储器时，才会出现数据的丢失。

  如果备用数据库在将日志记录从存储器复制到磁盘之前崩溃，则备用数据库上将丢失日志记录。通常，当备用数据库重新启动时，它可以从主数据库中获取丢失的日志记录。然而，如果主数据库或网络上的故障使检索无法进行，并且需要故障转移时，日志记录将不会出现在备用数据库上，而且与这些日志记录相关联的事务将不会出现在备用数据库上。

  如果事务丢失，则在故障转移操作之后，新的主数据库与原始主数据库不相同。客户机应用程序应该考虑重新提交这些事务，以便使应用程序状态保持最新。

  当主数据库和备用数据库处于对等状态时，如果主数据库发生故障，则在没有使用完全复原操作重新初始化的情况下，原始主数据库可能无法作为备用数据库重新加入 HADR 对。

  如果故障转移涉及丢失的日志记录(因为主数据库和备用数据库已发生故障)，主数据库和备用数据库上的日志顺序将会不同，并且在未执行复原操作的情况下，重新启动原始主数据库以作为备用数据库的尝试将会失败。如果原始主数据库成功地重新加入 HADR 对，则可以通过发出未指定 BY FORCE 选项的 TAKEOVER HADR 命令来完成数据库的故障恢复。

  如果原始主数据库无法重新加入 HADR 对，则可以通过复原新的主数据库的备份映像来将其重新初始化为备用数据库。



局域网环境一般采用 NEARSYNC 方式进行同步。

- ASYNC(异步)



如果主系统发生故障，此方式发生事务丢失的几率最高。在三种方式之中，此方式的事务响应时间也是最短的。

在此方式中，只有当日志记录已写入主数据库上的日志文件，而且已将此记录传递给主系统主机的 TCP 层时，才认为日志写入是成功的。因为主系统不会等待来自备用系统的应答，所以当事务仍处于正在传入备用系统的过程中时，可能会认为事务已落实。

主数据库主机上、网络上或备用数据库上的故障可能导致传送中的日志文件丢失。如果主数据库可用，则会在此对重新建立连接时，将丢失的日志文件重新发送至备用数据库。然而，如果在丢失日志文件时要求执行故障转移操作，则日志文件和相关联的事务都将不会到达备用数据库。丢失的日志记录和主数据库上的故障会导致事务的永久丢失。

如果事务丢失，则在故障转移操作之后，新的主数据库与原始主数据库不是完全相同的。客户机应用程序应该考虑重新提交这些事务，以便使应用程序状态保持最新。

当主数据库和备用数据库处于对等状态时，如果主数据库发生故障，则在没有使用完全复原操作重新初始化的情况下，原始主数据库可能无法作为备用数据库重新加入 HADR 对。

如果故障转移涉及丢失的日志记录，主数据库和备用数据库上的日志顺序将会不同，并且重新启动原始主数据库以作为备用数据库的尝试将失败。因为，如果在异步方式中发生故障转移，日志记录更有可能丢失，所以主数据库不能重新加入 HADR 对的可能性也更大。如果原始主数据库成功地重新加入 HADR 对，则可以通过发出未指定 BY FORCE 选项的 TAKEOVER HADR 命令来完成数据库的故障恢复。如果原始主数据库无法重新加入 HADR 对，则可以通过复原新的主数据库的备份映像将此数据库重新初始化为备用数据库。

2. BBP 系统结构

某企业目前有一套物流 BBP 系统，基于 SAP 构建，SAP 后台数据库使用的是 DB2 v8.2。现有一台闲置服务器(IP：192.168.100.23)，使用这台服务器作为 HADR 系统的备份节点，已经上线运行的 BBP 服务器(IP：192.168.100.231)作为系统的主节点。B2B 系统目前的拓扑与实现 HADR 之后的拓扑图如图 17.4 所示。

HADR 系统将目前闲置的备份机充分利用了起来，一旦主节点发生故障，可以立即手动切换到备份节点，与此同时，客户端会自动重新连接到备份机，使得生产继续进行。故障的主机可以离线进行故障恢复。恢复之后可以加入 HADR 组，并且重新接管所有应用。表 17.2 简要说明了实施 HADR 的过程。

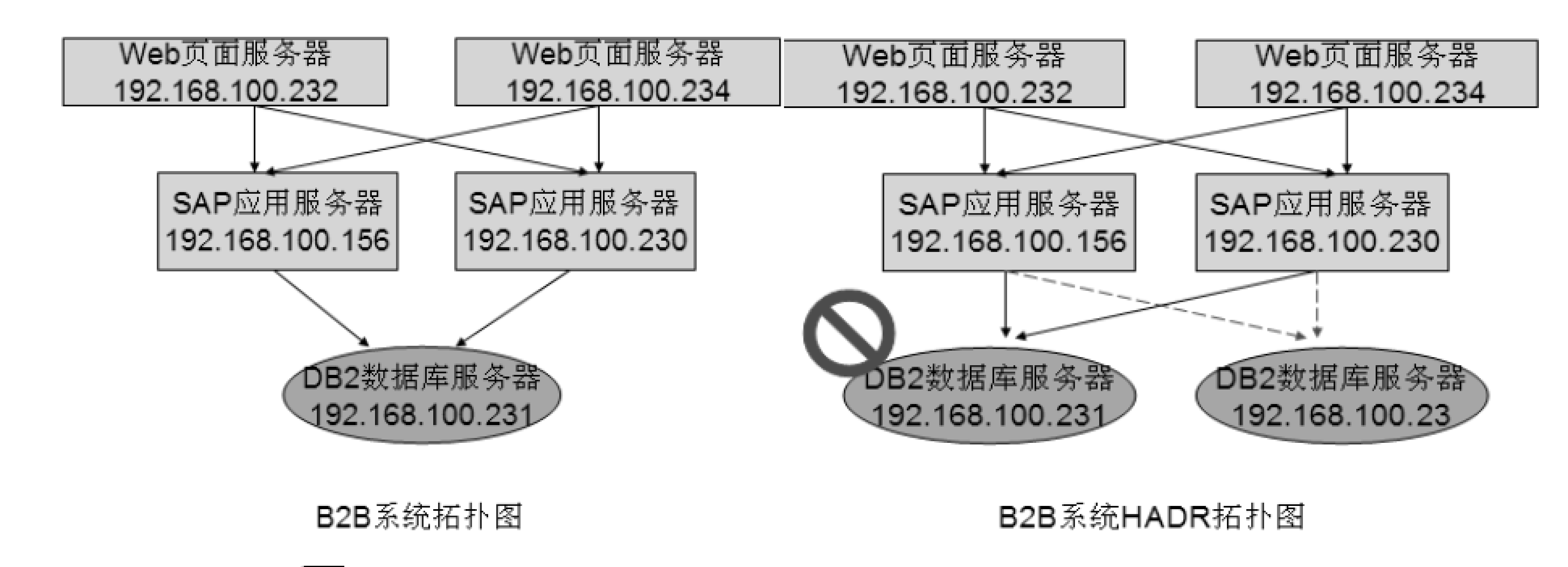


图17.4 B2B 系统目前的拓扑与实现 HADR 之后的拓扑图

表 17.2 在上述环境下实施 HADR 的简单过程

序号	所做操作	预计消耗时间(分钟)
1	对主机数据库做离线备份，数据量 75GB。 拷贝主机数据库的离线备份镜像到备份机	220
2	检查主机和备份机的磁盘分区、数据库管理器配置参数、数据库配置参数、活动日志路径、归档日志路径、容器路径、用户名和密码、环境变量，如有不同，修改备份机配置与主机一致	20
3	用镜像恢复备份机上的数据库	150
4	再次检查恢复之后数据库的配置与主机是否一致	5
5	在主机和备份机上分别用写好的脚本配置 HADR 所需的参数(脚本见附录)。 主机执行的脚本名：hadr_pri.bat 备份机执行的脚本名：hadr_std.bat	2
6	启动备份机上的 HADR： Db2 “deactivate db bbp” Db2 “start hadr on db bbp as standby”	1
7	启动主机上的 HADR： Db2 “deactivate db bbp” Db2 “start hadr on db bbp as primary”	1
8	检查主机和备份机 HADR 的运行状态： Db2pd –db bbp –hadr Db2 “get snapshot for db on bbp” 等待两边 HADR 处于对等状态	1
9	Takeover 测试。 先停止三台应用服务器上的 SAP。 备份机上执行： Db2 “takeover hadr on db bbp” Db2 “get snapshot for db on bbp” 检查是否备份机成功接管了主机的角色。 在备份机上创建测试表： Db2 “create table bbpadm.hadrtest(a int)” Db2 “insert into bbpadm.hadrtest values(100)” 在主机上执行： Db2 “takeover hadr on db bbp” Db2 “select * from bbpadm.hadrtest” Db2 “drop table bbpadm.hadrtest” 检查备份机上创建的表是否已经同步到主机	1

续表

序号	所做操作	预计消耗时间(分钟)
10	Takeover by force 测试 在备份机上执行： Db2 “takeover hadr on db bbp” Db2 “get snapshot for db on bbp” 在主机上执行： Db2 “takeover hadr on db bbp by force” Db2 “get snapshot for db on bbp” 此时 HADR 应该处于断开状态。 Db2 “connect to bbp” 从强制接管恢复到 HADR 配对的步骤。 在主机上执行： Db2 “stop hadr on db bbp” 在备份机上执行： Db2rfpen on bbp Db2 “terminate”; db2stop Db2start Db2 “start hadr on db bbp as standby” 在主机上执行： Db2 “start hadr on db bbp as primary” Db2 “get snapshot for db on bbp” 检查 HADR 是否重新配对	10
11	客户端自动重路由测试 在 192.168.100.230 和 192.168.100.156 上执行： Db2 “disconnect bbp user 用户名 using 密码” Db2 “connect to bbp” Db2 “list db directory” 查看是否已经收到备份机地址和端口。 在 HADR 备份机上执行： Db2 “takeover hadr on db bbp” Db2 “get snapshot for db on bbp” 在 192.168.100.230 和 192.168.100.156 上执行: Db2 “disconnect bbp” Db2 “connect to bbp user 用户名 using 密码” Db2 “list db directory” 检查是否收到了另一台 DB2 服务器的地址和端口。 回切 HADR 角色 主机上执行： Db2 “takeover hadr on db bbp” Db2 “get snapshot for db on bbp” 在 192.168.100.230 和 192.168.100.156 上执行: Db2 “disconnect bbp” Db2 “connect to bbp user 用户名 using 密码” Db2 “list db directory” 检查是否收到了备用 DB2 服务器的地址和端口	10

- Hadr_pri.bat 内容

```
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_HOST 192.168.100.231"
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_SVC 64000"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_HOST 192.168.100.23"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_SVC 64001"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_INST db2bbp"
db2 "UPDATE DB CFG FOR bbp USING HADR_SYNCMODE NEARSYNC"
db2 "UPDATE DB CFG FOR bbp USING HADR_TIMEOUT 60"
db2 "update alternate server for db bbp using hostname 192.168.100.23
port 5912"
db2 "update db cfg for bbp using logindexbuild on"
db2 "update db cfg for bbp using indexrec restart"
```

- hadr_std.bat 内容

```
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_HOST 192.168.100.23"
db2 "UPDATE DB CFG FOR bbp USING HADR_LOCAL_SVC 64001"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_HOST 192.168.100.231"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_SVC 64000"
db2 "UPDATE DB CFG FOR bbp USING HADR_REMOTE_INST db2bbp"
db2 "UPDATE DB CFG FOR bbp USING HADR_SYNCMODE NEARSYNC"
db2 "UPDATE DB CFG FOR bbp USING HADR_TIMEOUT 60"
db2 "update alternate server for db bbp using hostname 192.168.100.231
port 5912"
db2 "update db cfg for bbp using logindexbuild on"
db2 "update db cfg for bbp using indexrec restart"
```

完成配置之后，在主机或者备份机上的 DB2 命令行环境中输入以下命令：

```
Db2 get snapshot for db on bbp
```

即可以查看 HADR 的运行状态，如图 17.5 所示。

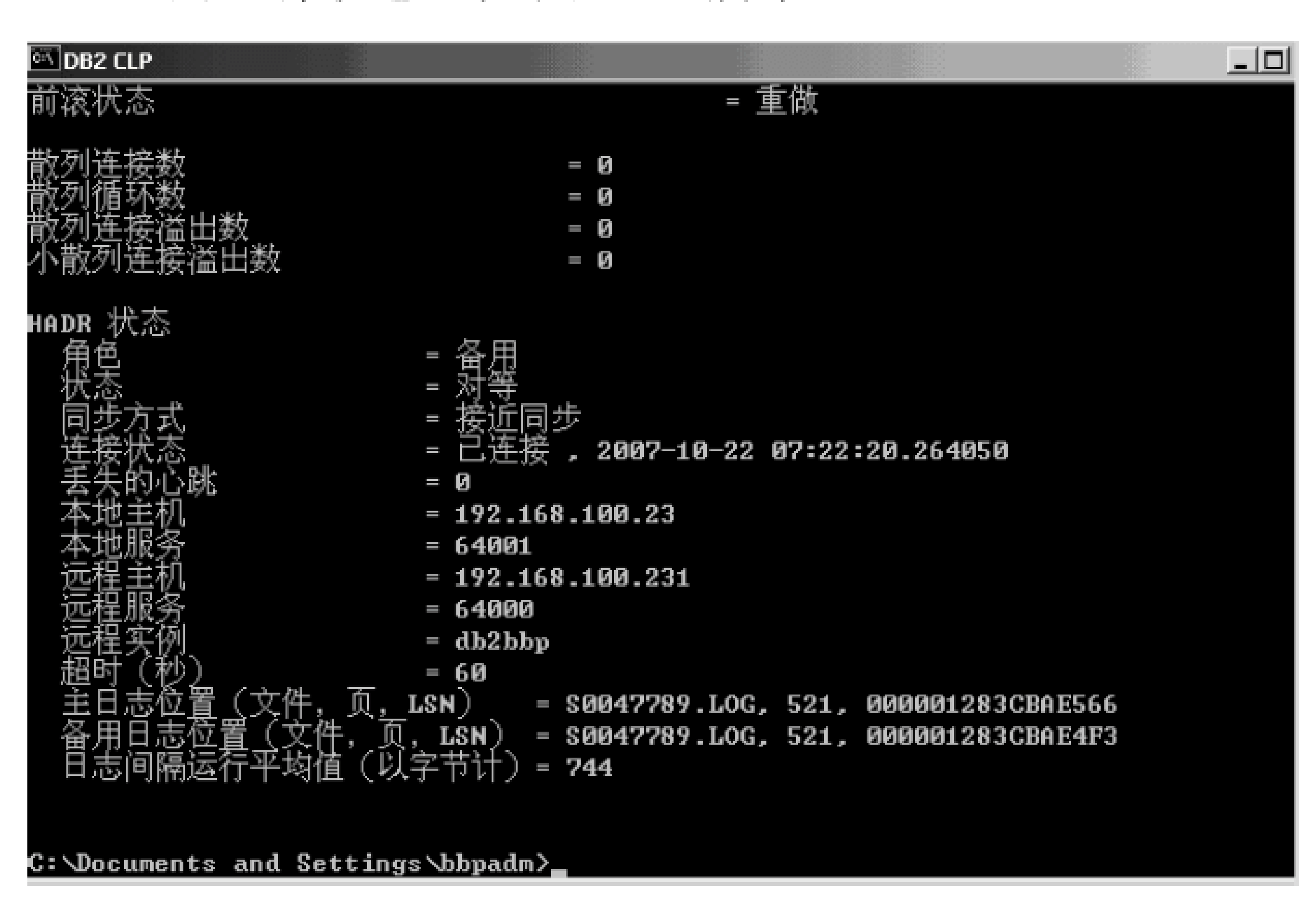


图 17.5　HADR 的运行状态

17.2.3　通过主机软件实现后端专用网络同步

用这种方式来同步数据，数据不会流经前端网络，而全部通过后端网络传输到备份站点对应的存储设备中。这就需要将主站点的后端网络设施和备份站点的后端网络设施连接起来。或者直接通过裸光纤连接两台 SAN 交换机；再或者租用电信部门的光缆专线。

如果用前者连接两个站点，那要求两个站点之间的道路上可以自己布线，比如一个大

院内，可以自主布线，不需要经过市政部门干预，这样便可以直接连接两端的 SAN 交换机，直接承载 FC 协议了。否则，如果两个站点跨越了很远的距离，那么就必须使用后者，也就是租用电信部门的光缆，但是这条光缆上的数据就必须符合电信部门传输设备所使用的协议了。后者需要添加额外的协议转换设备，两个站点各一个，如图 17.6 所示。

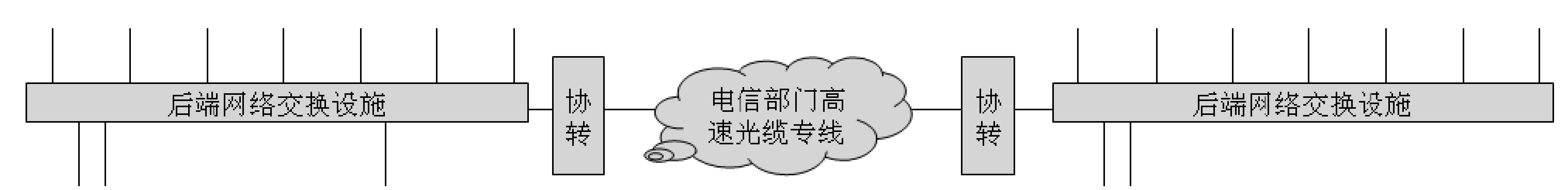


图17.6　连接两个站点的后端网络

现在电信部门的光纤专线一般为 SDH 传输方式，接入到用户端的时候，一般将信号调制成 E1、OC3 等编码方式，所以必须将 FC 协议承载于这些协议之上，也就是我们在前面第 13 章中所描述的 Protocol Over Protocol 模型，完成这个动作的，就是协议转换器。

协议转换器在一端按照某种协议的逻辑进行工作，而在另一端则按照另一种协议的逻辑进行工作，把数据从一端接收过来，经过协议转换，以另一种协议的逻辑发送出去，到达对端后，再进行相反的动作。

有些路由器则直接在其内部集成了各种协议转换器，可以说路由器就是一种协议转换器。我们可以看一下机房中的网络路由器，上面有各种各样的接口，为何不清一色都是 RJ45 以太网接口呢？

因为路由器不只是路由以太网数据，他还要路由其他网络协议的数据，甚至还要在不同网络协议之间做转换。而如果把这些协议都做到 SAN 交换机上，那么这台 SAN 交换机，就是一台不折不扣的 SAN 路由器了。对于没有费用购买 SAN 路由器的用户来说，用一个层层的协议转换设备来完成也是很划算的，这就像给照相机加一层层的特殊镜头一样。图 17.7 是一个层层协议转换器的实例。

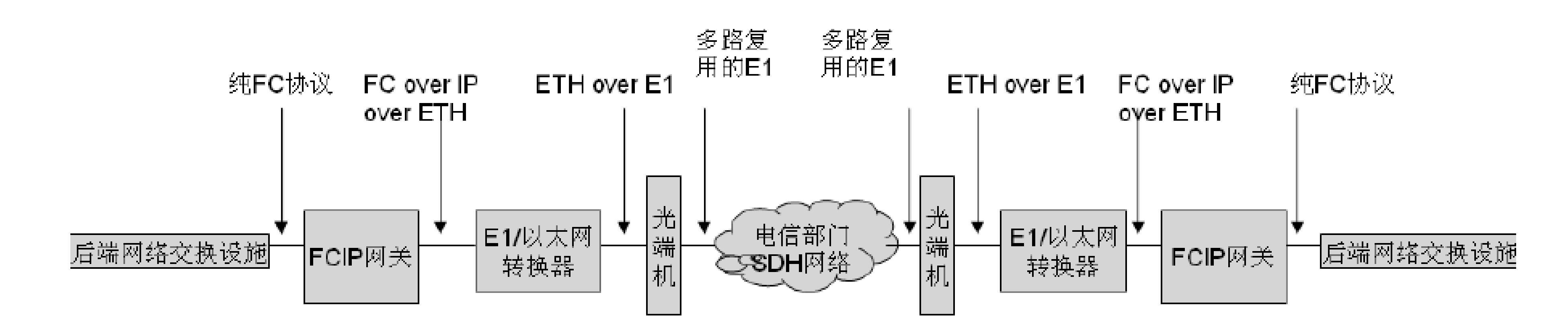


图17.7　利用 SDH 网络连接后端网络

源端的纯 FC 协议，经过 FCIP 网关，变成了基于以太网的 IP 协议(FC over IP over ETH)，经过 E1/以太网转换器，承载到了 E1 协议之上，然后多路 E1 信号汇聚到光端机，通过一条或者几条光纤，传输给电信部门的 SDH 交换设施上进行传输，到达目的之后，进行相反的动作，最终转换成纯 FC 协议。这样，源和目的都不会感觉到中间一层层协议转换设备的存在。

主站点和备站点的后端 SAN 交换机能够成功连接之后，两个 SAN 网络便可以融合了，就像一个 SAN 网络一样。所以，主站点的服务器也就可以访问到了备份站点的磁盘阵列。这样，不需要经过前端网络，就可以直接访问备份站点的存储设备，也就可以直接在备份站点的存储设备上读写数据了。如图 17.8 所示，备份站点磁盘阵列上的一个 LUN(卷 B)可

以直接被主站点的服务器识别，这样，主站点的服务器就可以同时操作本地磁盘阵列和备站点的磁盘阵列了。

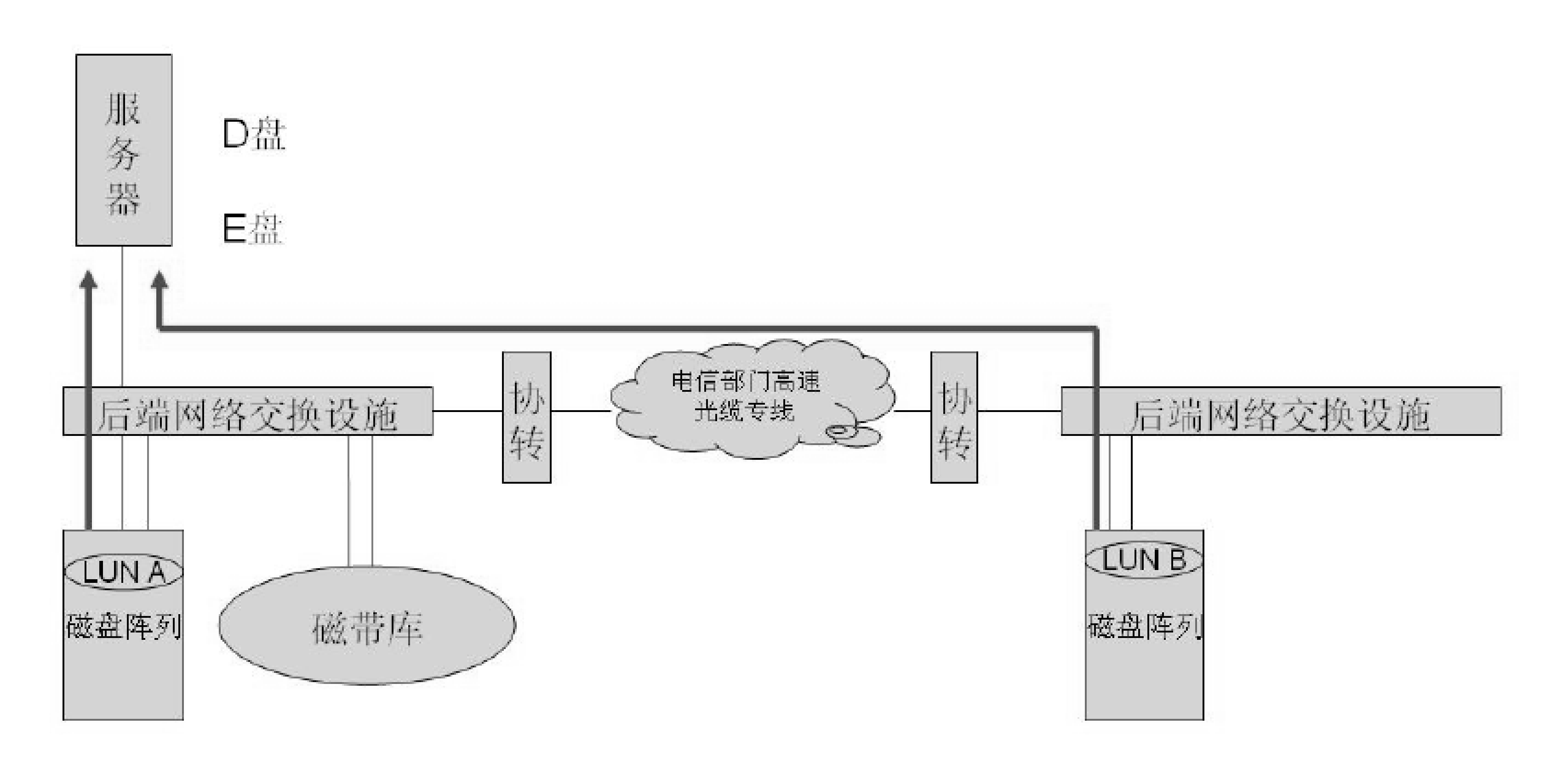


图17.8 后端网络通路示意图

我们来看一下这种方式下数据走过的路径。

本地磁盘阵列－SAN 网络交换设施－本地服务器内存－SAN 网络交换设施－通过协转流入电信部门网络(如果有)－远端 SAN 网络交换设施－远端磁盘阵列，如图 17.9 所示。



上述的两种方式中，第二种方式的步骤比第一种少了两步，数据到达远端 SAN 交换设施之后，立即被传送到了磁盘阵列这个最终目标，而不必再经过一台服务器了，为何呢？

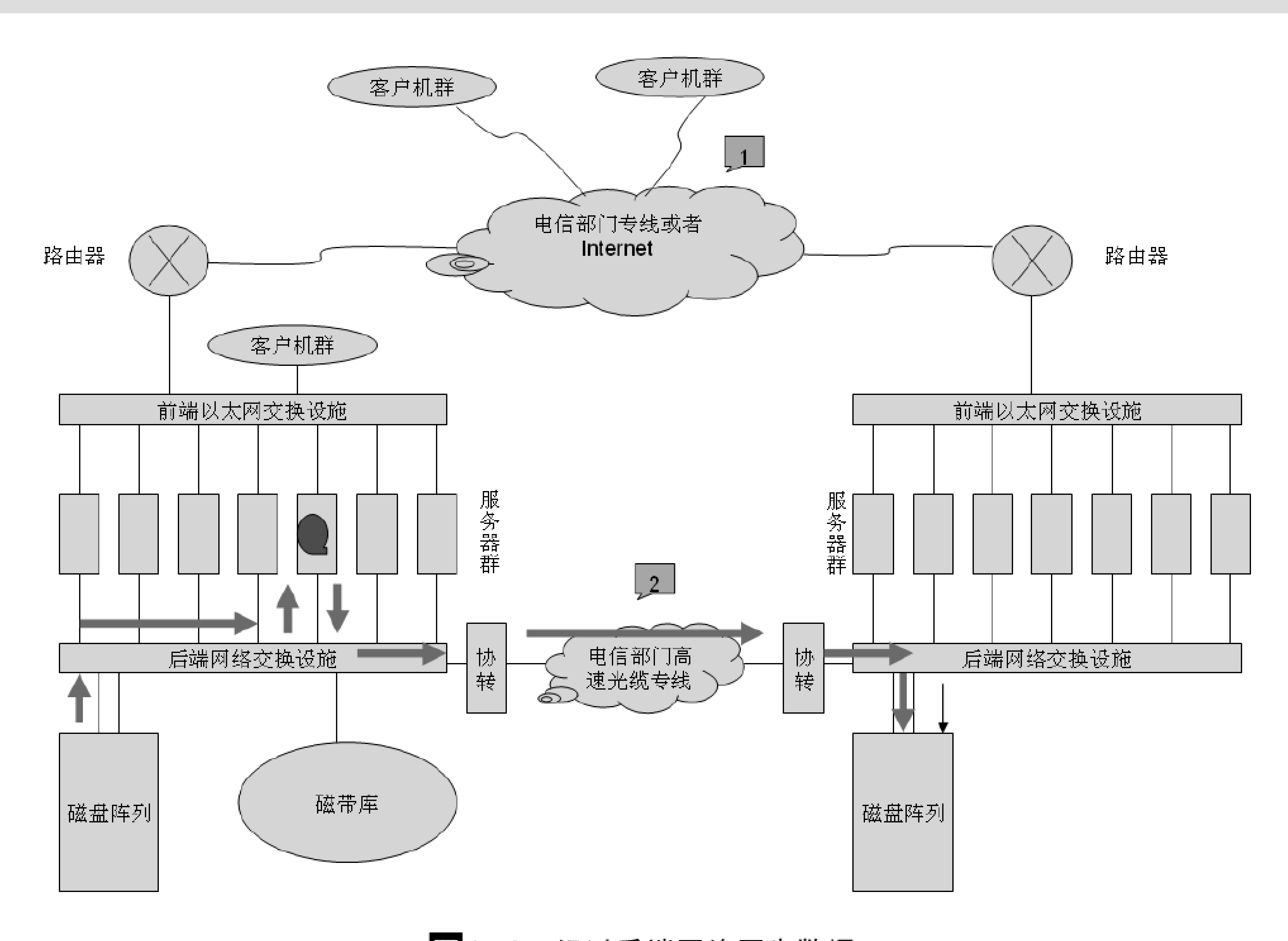


图17.9 经过后端网络同步数据

因为第二种方式中，主站点的服务器对备份站点的存储设备有了直接访问权，而第一种方式中，双方都没有对方存储设备的直接访问权，必须通过对方服务器的参与。

然而，第二种方式数据仍然至少需要经过一台服务器，为何呢？因为涡轮泵(实现数据同步功能的软件)是运行在服务器上的，没有涡轮泵，水就不会流动，数据也无法流动，而不可能有一种泵，可以让水不经过它就可以流动。同样，数据流也不可能不经过泵就自己流动。

现在，两个 SAN 已经连接了，而且主站点的服务器可以畅通无阻的访问备站点的磁盘阵列存储设备了，万事俱备，只欠东风。究竟在这种方式中，要怎么来设计这个涡轮泵呢？

其实没有什么特别指出，我们来分析第一种方式的同步方法，那个泵用的是从本地提取数据，发送到前端网络，网络那头用一个接收者将接收的数据写入到盘阵中。



同理，第二种方式下，大思路当然还是这样，只不过是从后端网络提取数据之后，再发送回后端网络的另一个目的，然而这一切只需要一个泵就能完成，因为这个泵现在已经可以掌管数据的起源设备和数据的终结设备了。

我们理所当然的设计了这个泵，它的作用方式就是，将数据从本地的卷 A 中提取出来，然后直接通过 SAN 网络写入位于备份站点的卷 B。如果数据是直接在内存中生成的，需要写入保存，则写入本地卷 A 一份，同时写入远端的卷 B 一份。这种方式显然比第一种方式来的快，但是它对网络速度要求更高，成本也更高。第一种方式中，有两个泵，而第二种方式中，只有一个泵，这样会不会造成“动力”不足呢？不会的，水在流动过程中是有阻力的，而数据流是没有阻力的，所以如果增加额外的泵，反而会影响数据流的速度。

这种实现方式又叫做“卷镜像”，意思就是两个卷像镜子一样完全一样。第一种方式为何不叫镜像呢？因为第一种方式跨越的距离太远，这样不能达到两个卷在任何时刻的数据都相同，在讲同步和异步的时候还会涉及这方面的问题。这种方式能很好的保证数据同步的实时性，但是不适合远距离大数据量数据同步，除非不惜成本搭建高速远距离专线链路。

第二种方式，卷同步软件是工作在卷这一层的，所以它检测的是数据块的变化而不是文件的变化，同步的数据内容是数据块而不是文件，和第一种方式有所不同。

1. Veritas Volume Manager 软件介绍

Veritas Volume Manager 也是 Veritas 公司 Storage Foundation 套件中的一个模块，它的功能就是辅助或者代替操作系统自己的磁盘管理模块来管理底层的物理磁盘(当然也有可能是 SAN 上的 LUN 逻辑磁盘)。

一般操作系统自己的磁盘管理模块功能有限，比如 Windows 提供的磁盘管理器组件，其功能只限于对识别到的物理磁盘进行分区、格式化、挂载到某个盘符下，并且分区只能连续，而且不能动态调整分区大小，要调整也只能删除分区再重新建立，当然一些第三方软件可以做到调整分区大小，不过仍然需要重启。Windows Server 操作系统提供的动态磁盘管理，虽然可以做到 RAID 卡的部分功能，但是仍然不够灵活，而且效率低下。而 VxVM(Veritas Volume Manager)卷管理软件可以彻底替换操作系统的卷管理功能。

> 提示：本书第 5 章中曾经通俗的阐释了卷管理软件的思想，VxVM 就是这样一个卷管理软件，它把操作系统底层的磁盘统统当作“面团”，可以糅合起来，然后再分配。

它改变了操作系统的磁盘管理器的分区管理的闲置，将所有磁盘虚拟成卷池，然后从池中分配新的卷，新卷可以动态的增大和减小容量，可以动态分割、合并。支持卷多重镜像。支持 RAID 0，RAID 1， RAID 0+1，RAID 5。支持 RAID 组在线动态扩容，可随时向 RAID 组中添加新磁盘而不影响使用。卷之上还需要有一层文件系统，VxVM 同样有自己的文件系统，叫做 VxFS。VxFS 是一个高效的日志型文件系统，这就使得在发生崩溃之后文件系统的自检过程非常快。另外，还支持文件系统大小动态扩充和收缩。支持 Direct IO。支持文件系统快照功能。

用户只要在服务器上安装 VxVM 软件，经过相关配置，就可以实现对服务器上两个卷的镜像操作，实现两个卷的数据同步。一旦某时刻主站点发生故障，则备站点的卷上数据和主站点发生故障的时候完全一致。此时只要在备份站点的服务器上挂载这个卷到某个盘符(windows)或者目录(UNIX)下，便可以继续使用了。

2. Logcal Volume Manager 软件介绍

Logcal Volume Manager(LVM)是 Linux 系统上的一个开源的软件，后来被 IBM 的 AIX 操作系统用于默认的卷管理模块。LVM 相对于 VxVM 来说，是一个更加开放、通用的卷管理软件。LVM 同样也可以对两个卷进行镜像操作。在本书第 5 章已经介绍过 LVM，这里就不做过多描述了。

17.2.4 通过数据存储设备软件实现专用网络同步

在前两种方式中，描述过数据要流动，就需要一个泵来提供动力。第一种方式中，有两个泵，数据流经的管道最长；第二种方式中，有一个泵，数据流经的管道比第一种要短。这两种方式，泵都被安装在了服务器上。而第三种方式，泵没有安装在服务器上，也没有安装在网络设备上，而是被安装在了存储设备上，如图 17.10 所示。

数据最终还是要存储在存储设备上，与其让别人从自己身上提取数据然后发送到远端，不如自己动手，丰衣足食。第三种方式就是利用的这种思想，自己做主将自己的数据通过后端 SAN 网络设施传输到目标设备上。

如图所示，主站点的磁盘阵列设备上的同步软件，从自身的一个卷(LUN A)提取数据，通过 SAN 交换机传输给了备份站点的磁盘阵列设备上的同步软件接收端，并将接收到的数据写入镜像卷(LUN B)。

数据流的路径如下：本地磁盘阵列－本地 SAN 网络交换设施－电信部门交换机组(如果有)－远端 SAN 网络交换设施－远端磁盘阵列。

路径比第二种方式又少了两步。更加重要的是，这种方式彻底解脱了服务器，服务器上不需要增加任何额外的负担，所有工作全部有磁盘阵列设备自己完成。

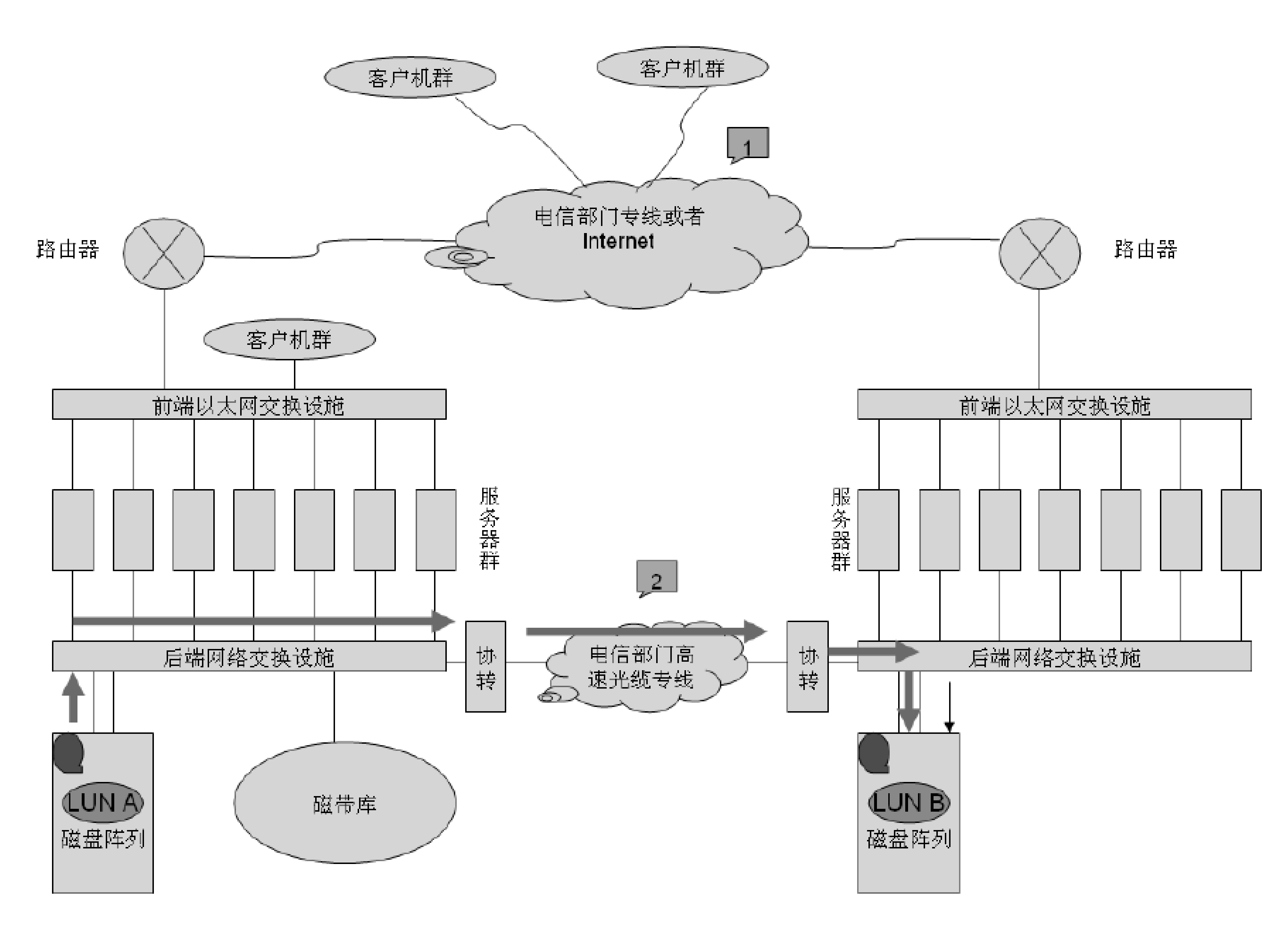


图17.10 经过后端网络同步数据

提示 在本书第 5 章详细讲解过磁盘阵列。磁盘阵列本身就是一个计算机系统，有自己的 CPU、RAM、ROM，甚至有自己的磁盘。磁盘阵列就是一台管理和虚拟化大量物理磁盘的主机系统。既然是主机系统，那当然可以在其上运行各种功能的软件了。所以，这种数据同步软件，应运而生。

提示 目前几乎每个厂家的高端磁盘阵列设备，都具有数据同步功能。比如 IBM 公司 DS 系列盘阵上的数据同步功能叫做 Remote Mirror。HDS 公司的叫做 TrueCopy，EMC 公司的叫做 SRDF，不管叫什么，它们的思想和原理都是一样的，只不过在实现方法和效果上有所不同。

这种方式的数据同步，由于底层存储设备不会识别卷上的文件系统，所以同步的是块而不是文件，也就是说存储系统只要发现某卷上的某个块变化了，就会把这个块复制到远程设备上。此外，备份站点的存储设备必须和主站点的存储设备型号一致，因为不同厂家的磁盘阵列产品之间无法做数据同步。而在主机上的同步引擎，就没有这种限制，因为主机上的同步软件所操作的是操作系统卷，而不是磁盘阵列上的卷，操作系统隐藏了底层存储阵列上的卷，不管什么厂家什么型号的盘阵，经过了操作系统的屏蔽，对应用程序看来，统统都是一个卷，或者一个盘符或目录。

17.2.5 案例：IBM 公司 Remote Mirror 容灾实施

Remote Mirror 是 IBM 公司的 DS4000 系列中端磁盘阵列上的一个软件模块，其功能就是将本地磁盘阵列上某个或者某些卷的数据，同步到远端磁盘阵列上对应的卷，支持同步

和异步复制，支持一致性组。

某企业存储系统如图 17.11 所示。

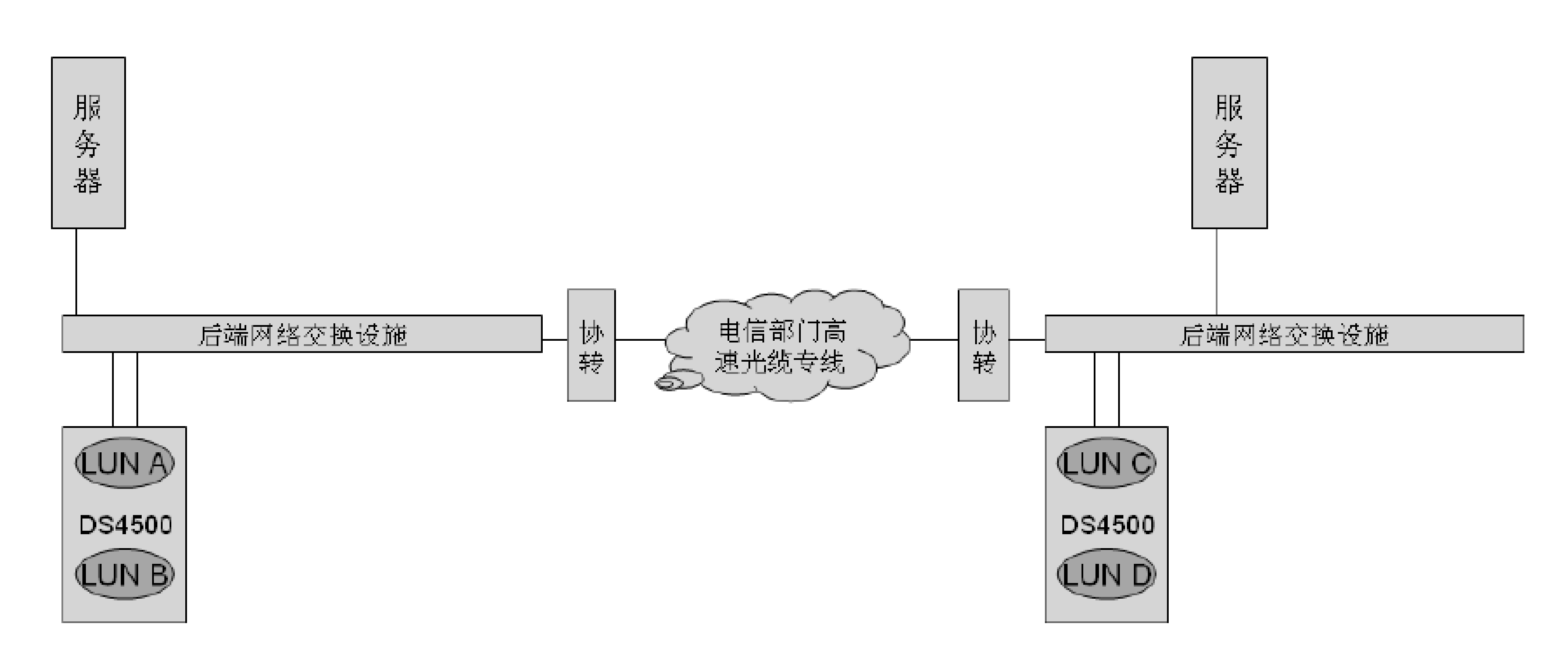


图17.11 某企业存储系统图

这个企业有两台 DS4500 磁盘阵列，主、备站点各一台。现在将主站点的两个卷 LUN A 和 LUN B 同步到备份站点的两个卷 LUN C 和 LUN D 上。此时需要启动 DS4500 的 Remote Mirror 功能。启用这个功能，要求主站点和备站点上必须分别建立一个用于数据缓冲以及相关重要数据存放的卷，且必须为镜像卷，大小 100MB 即可。

1】 在 Storage Manager 配置界面中，选择 Storage Subsystem | remote mirror | Active 菜单命令，选择在现有的 array 上建立一个 mirror 的 Repository 逻辑卷，如图 17.12 所示。

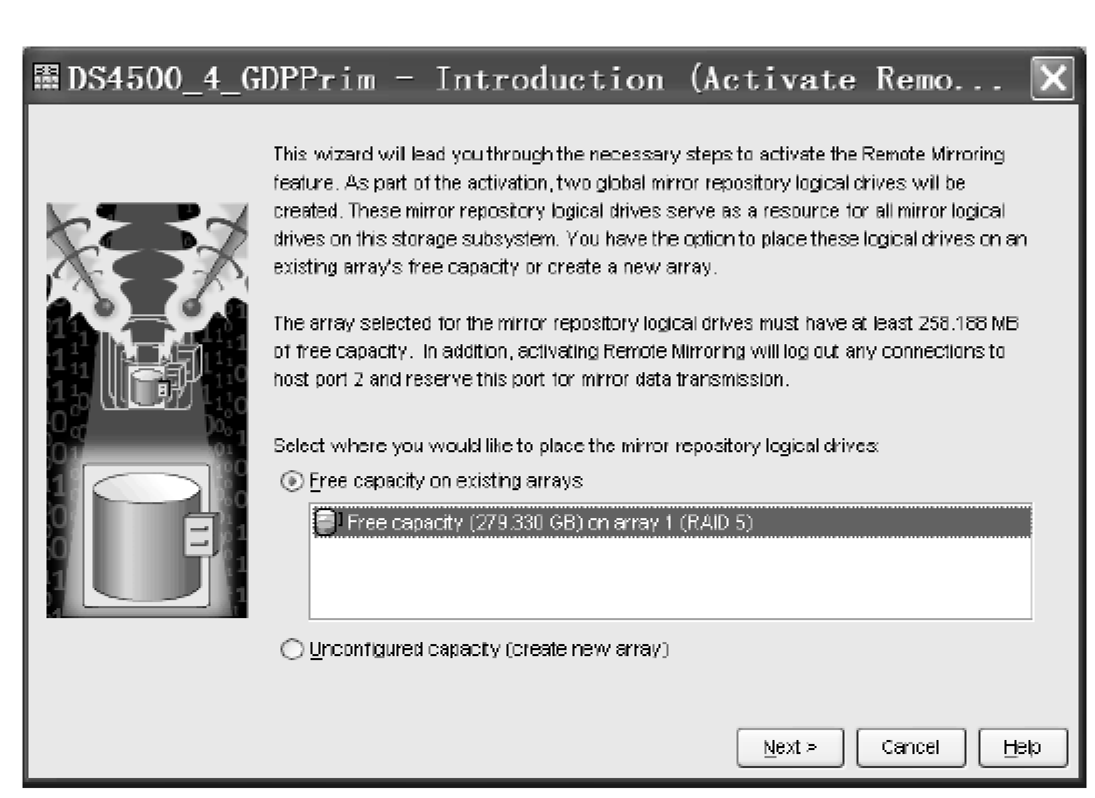


图17.12 创建 Repository 卷

2】 单击 Next 按钮，弹出如图 17.13 所示的对话框，提示 DS4500 磁盘阵列将前端的 2 号端口专门用于连接远程的磁盘阵列设备来传输数据，所以这个端口将禁止用于其他主机的连接。

3】 单击 Finish 按钮。提示在备份节点上也需要相同的操作，如图 17.14 所示。

4】 主站点上的两个 Repository 已经建立成功，如图 17.15 所示。

5】 在备份站点上重复上述步骤。然后，在备份站点磁盘阵列上建立两个目标卷，大小必须大于源卷，如图 17.16 所示。

6】 在主节点上右击 Create Remote Mirror，如图 17.17 和图 17.18 所示。

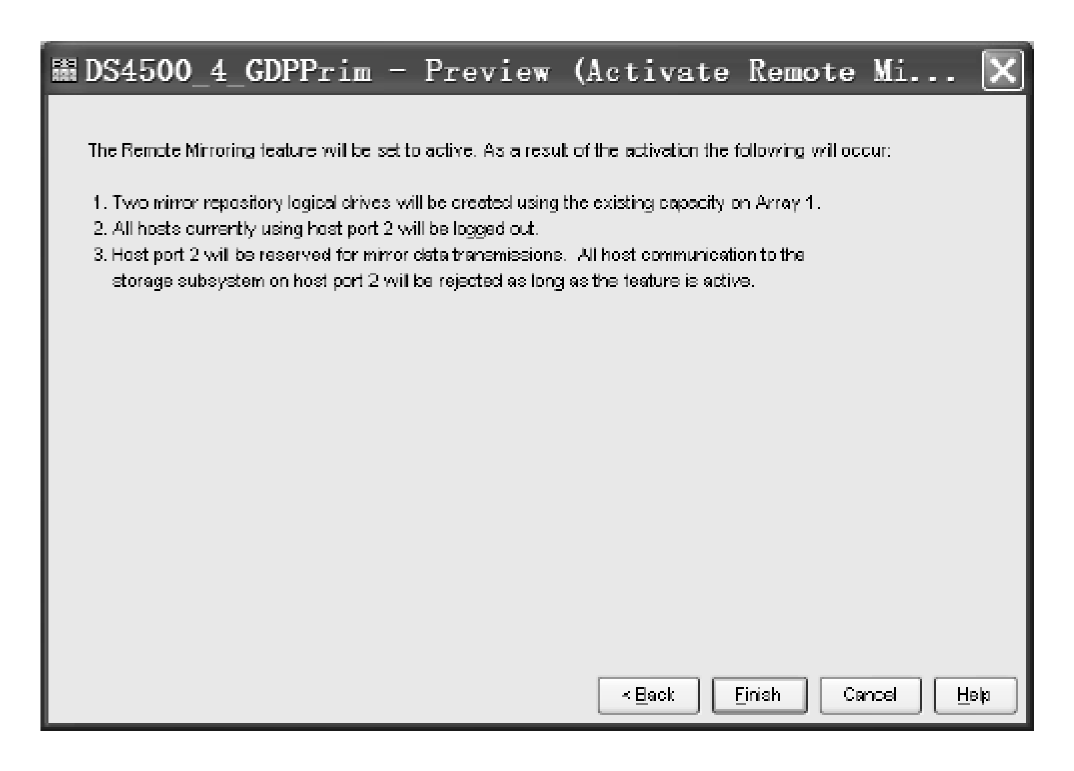


图17.13　重要提示窗口

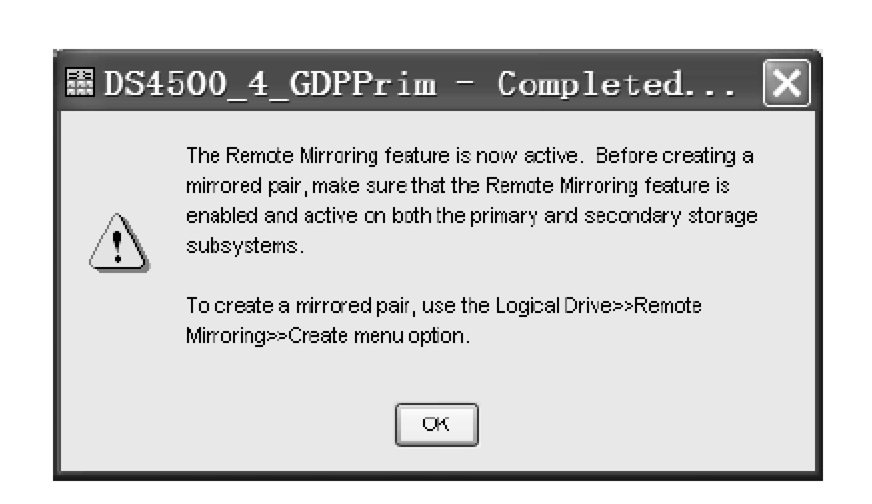


图17.14　提示备份节点需要相同的操作

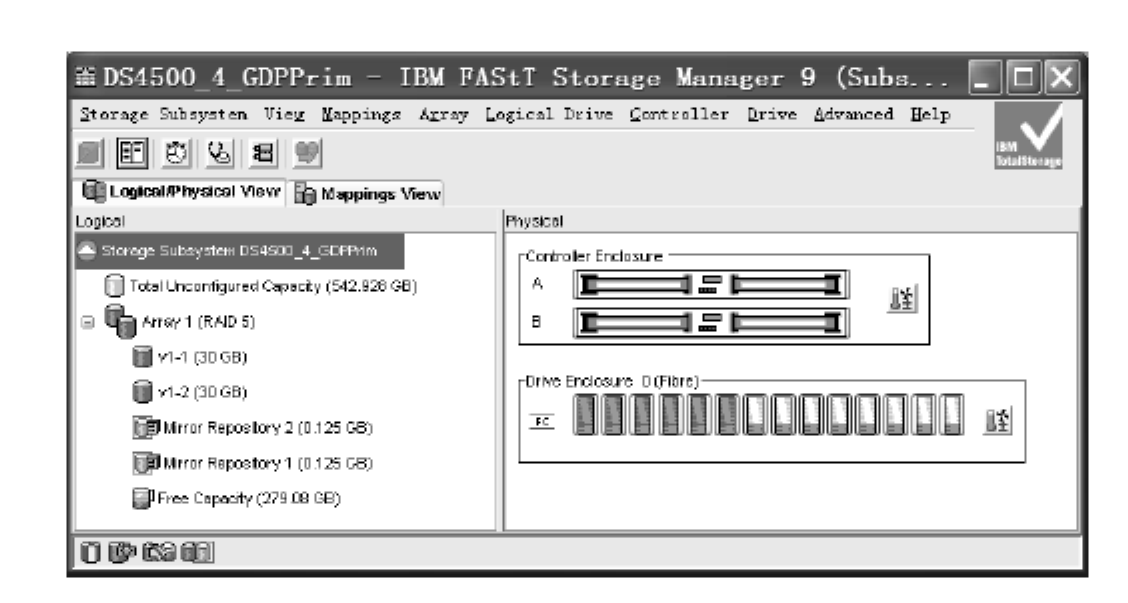


图17.15　成功建立 Repository 卷

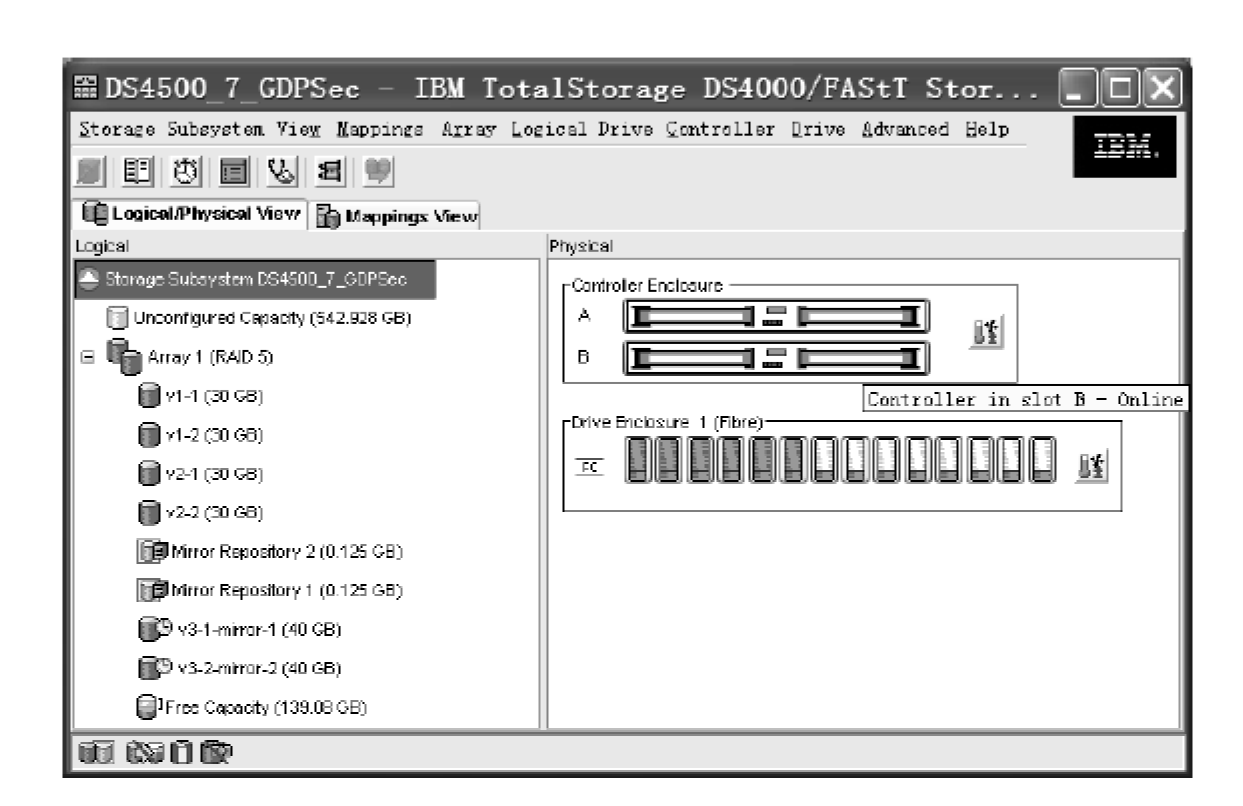


图17.16　建立两个目标卷

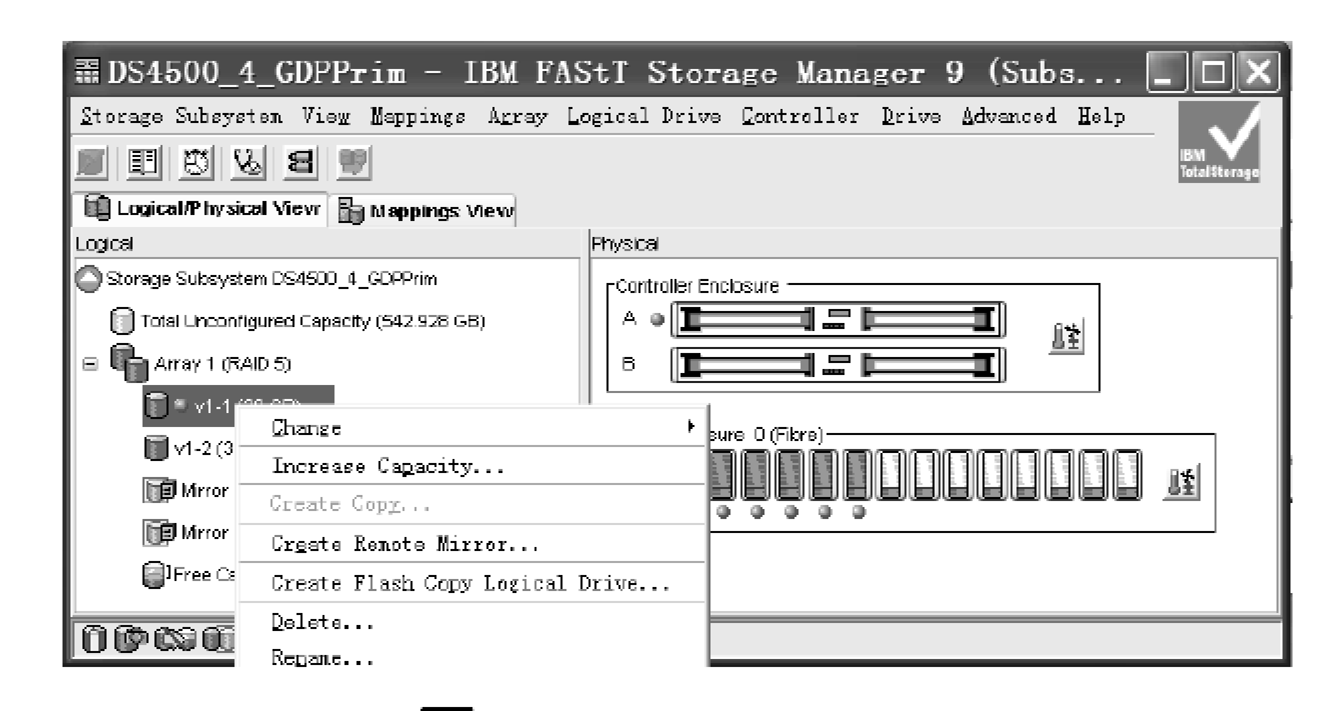


图17.17　创建远程镜像

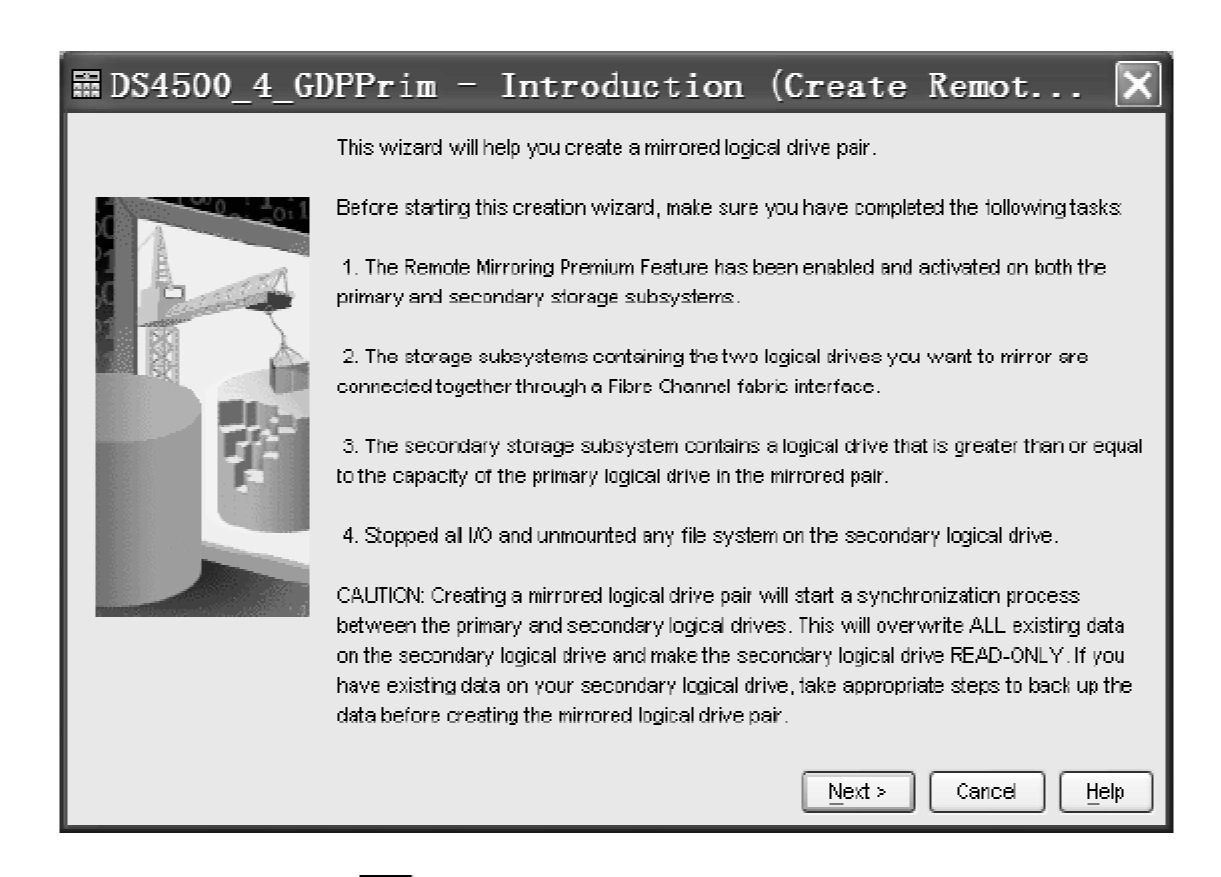


图17.18 创建远程镜像

此时，本地磁盘阵列会显示出网络上的另一台磁盘阵列设备，如图 17.19 所示。

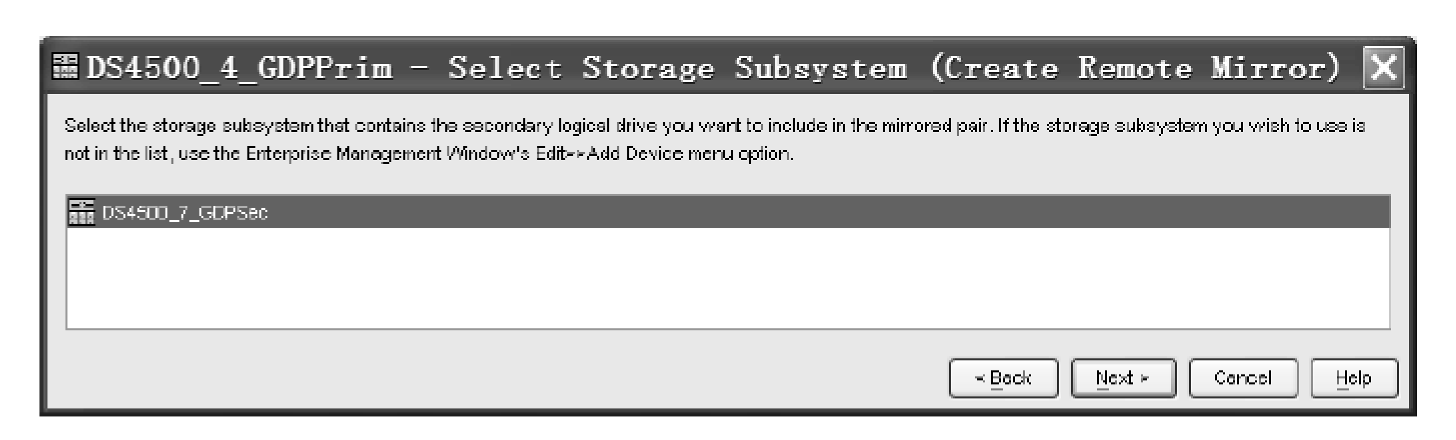


图17.19 远程设备列表

7】 选择备份磁盘阵列上的镜像目标盘，如图 17.20 所示。

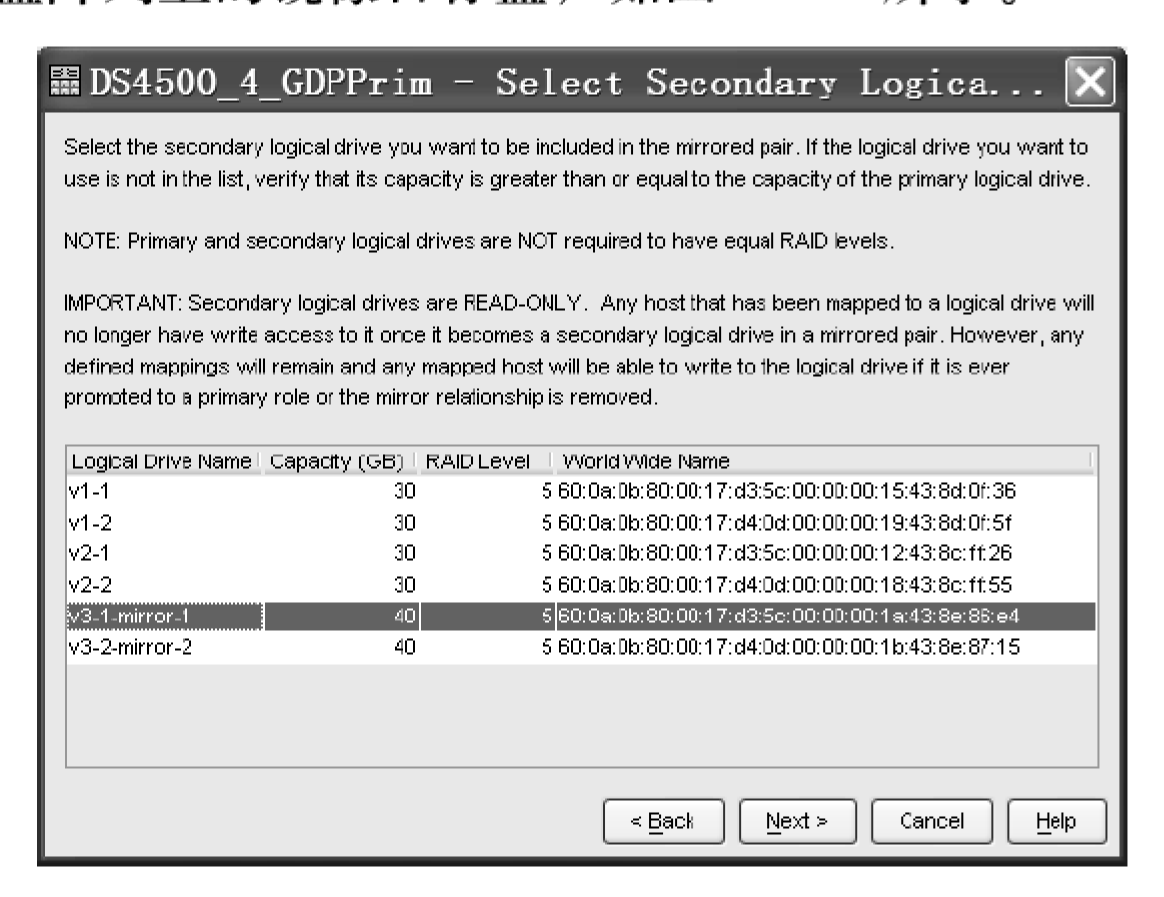


图17.20 远程设备上的卷列表

8】 选择模式，如图 17.21 所示。

9】 设置同步的优先级，如图 17.22 所示。

10】 输入“yes”以便确认操作，如图 17.23 所示。

11】 完成后提示如图 17.24 所示的对话框。

12】 用同样的方法作另外一个确认提示，如图 17.25 所示。

13】 完成后，监控主站点磁盘阵列的变化，如图 17.26 所示。

14】 监控备份站点磁盘阵列的变化，如图 17.27 所示。

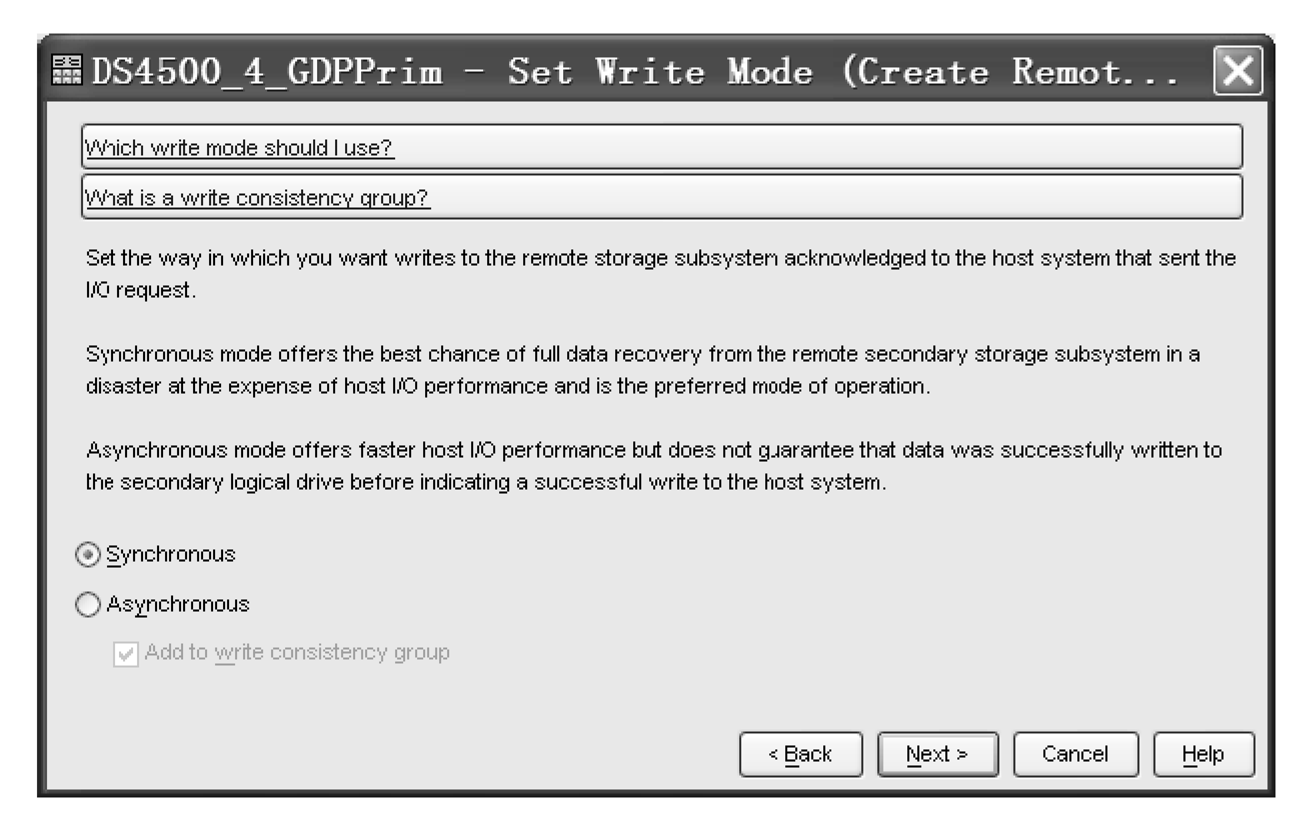


图17.21 同步或者异步模式选择

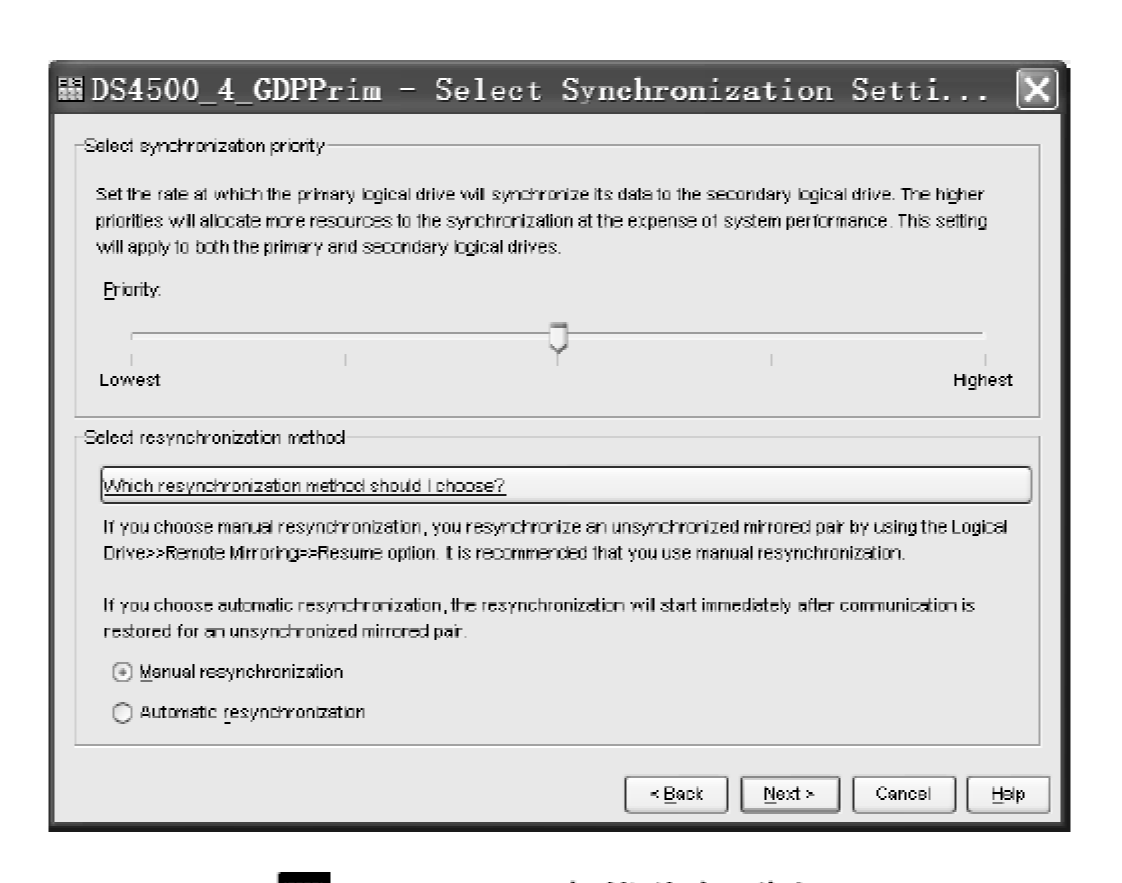


图17.22 同步优先级选择

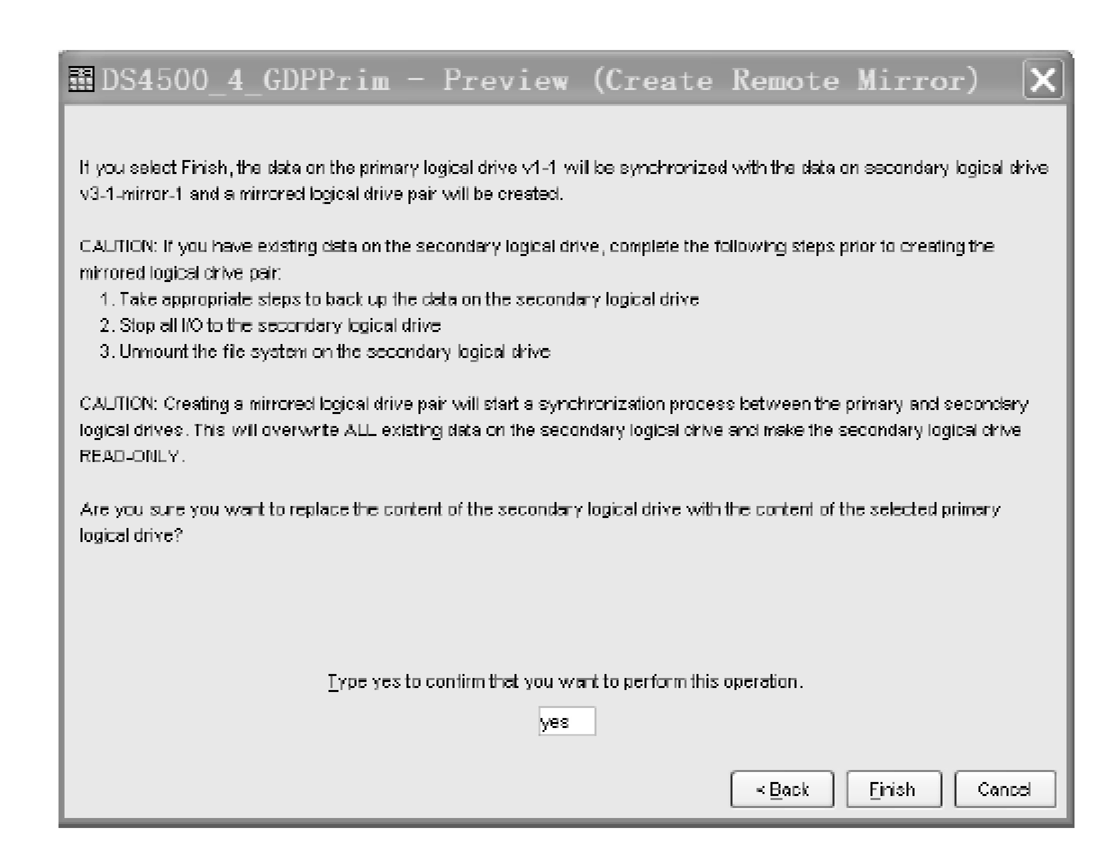


图17.23 确认

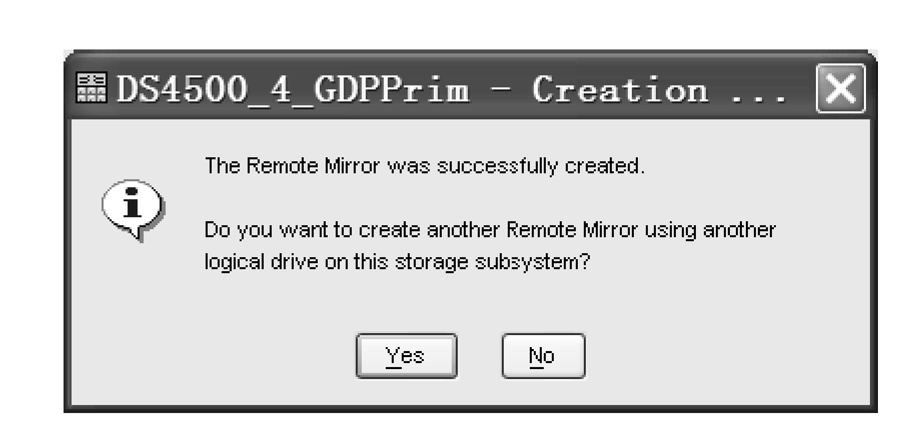


图17.24 成功提示

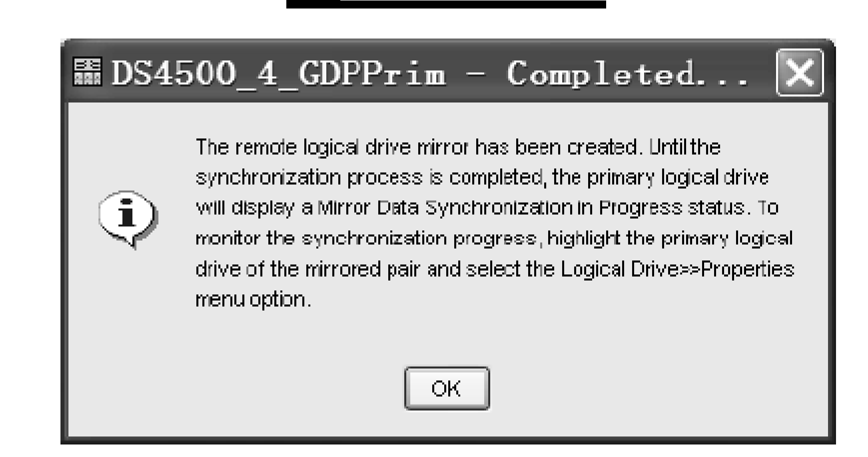


图17.25 确认提示

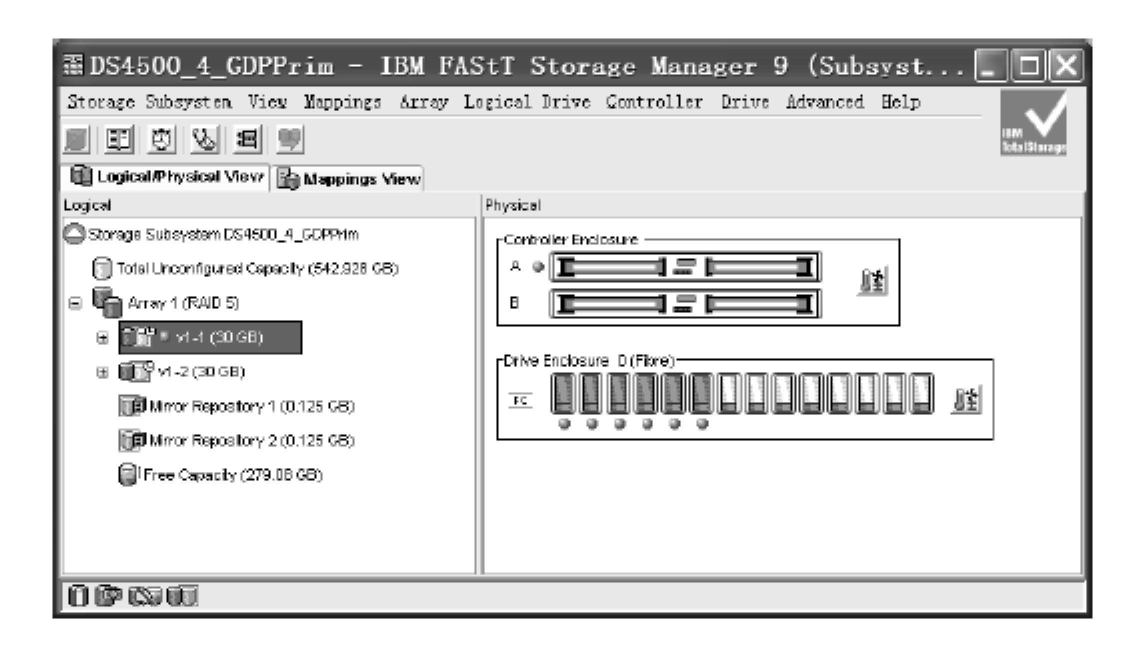


图17.26 镜像后卷图标的变化

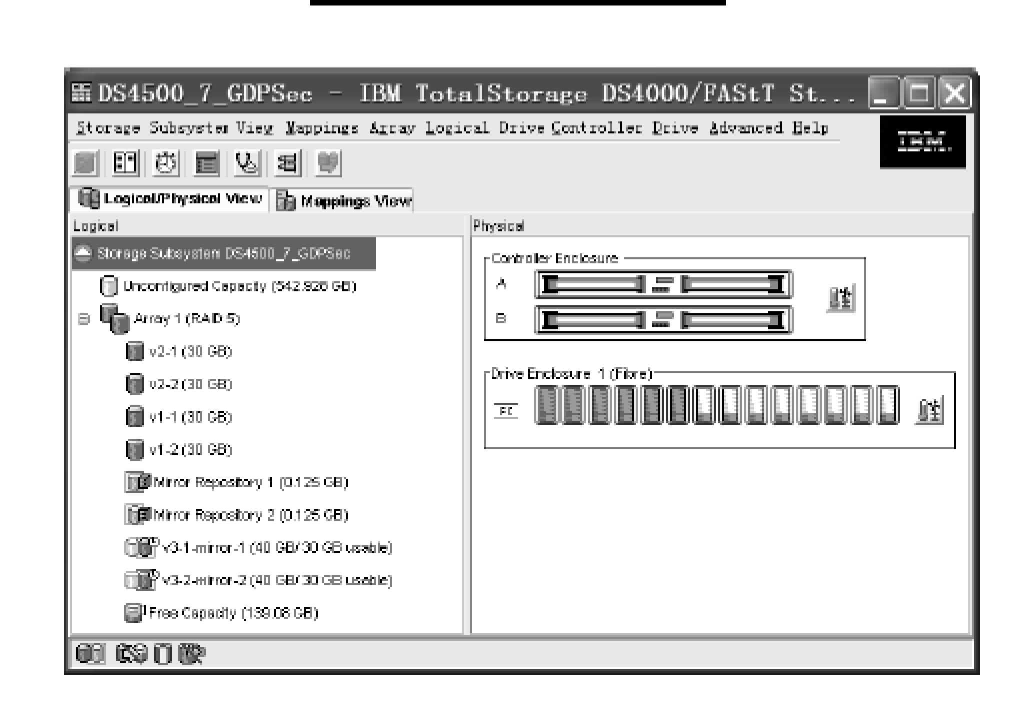


图17.27 备份站点图标的变化

主站上右击原盘/属性可以查看镜像的完成情况，如图 17.28 所示。

备份站方法相同，如图 17.29 所示。

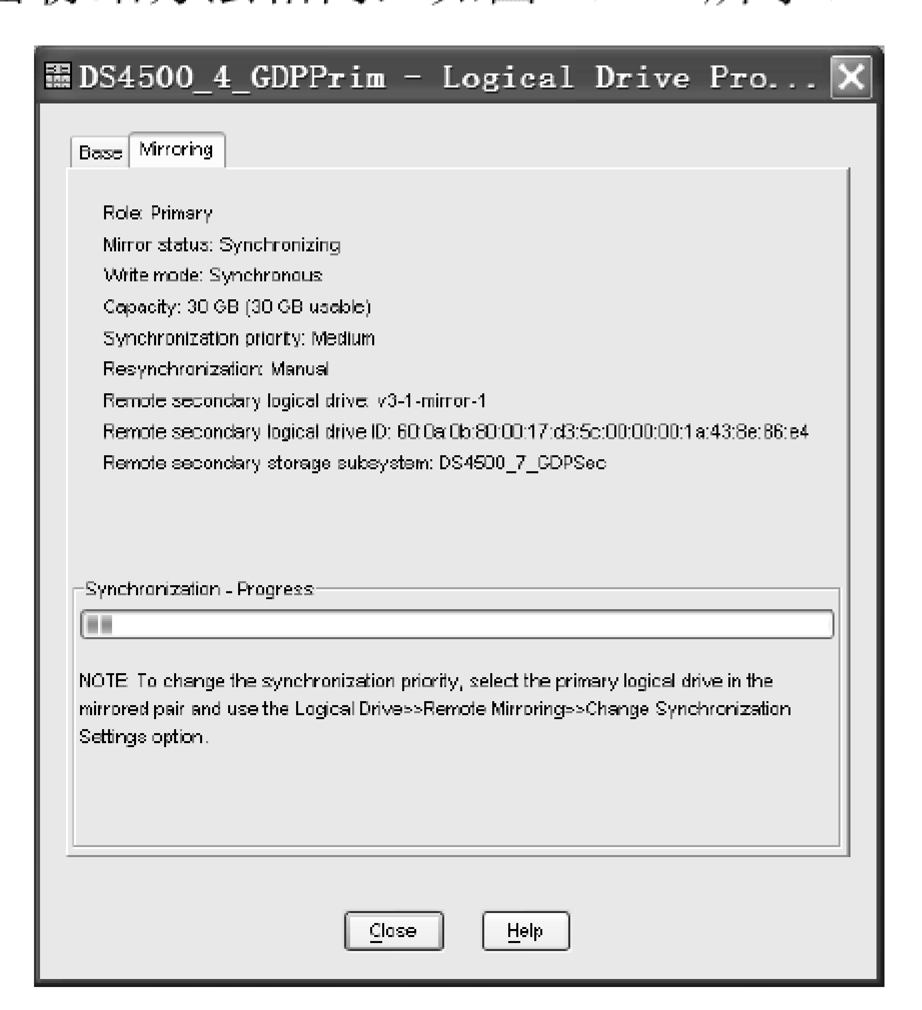


图17.28 镜像状态窗口

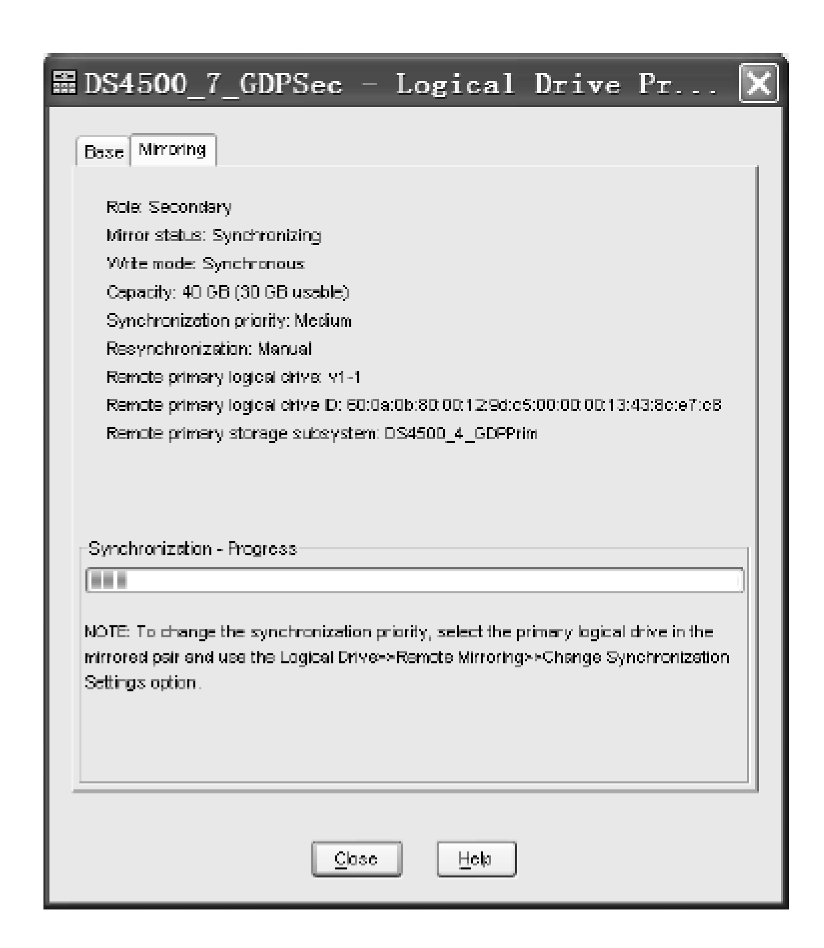


图17.29 备份站镜像状态窗口

17.2.6 小结

纵观以上三种数据同步的方式，可以发现。

第一种方式数据经过的路径最长，同步实时性最差，但是也最廉价。

第二种方式数据经过的路径适中，数据同步实时性强，但是对后端链路要求比第一种高，不适合大量数据同步。

第三种方式，数据经过的路径最短，对服务器性能没有影响，但是仍然不适合在远距离低速链路的环境下运行，而且还不能保证数据对应用程序的可用性，因为存储设备与应用程序之间还有操作系统这一层，操作系统有自己的缓存机制，如果存储设备上的数据同步引擎没有与操作系统配合良好的话，很有可能造成数据的不一致性，这样会影响到应用程序，甚至使应用程序崩溃。

主站点发生故障之后，备份站点的存储设备会感知到，然后强行接管主站点的工作，断开同步连接，备份站点成为当前的主站点，接受应用程序的读写操作，并记录自从断开连接之后发生变化的数据块。待探测到主站点恢复正常之后(或者手动重新配置同步)，备份站点先将变化的数据块复制回原来的主站点，复制完成后，原来的主站点再次接管回主角色，成为当前的主站点，接受应用程序的读写操作。

17.3 容灾中数据的同步复制和异步复制

17.3.1 同步复制例解

下面来分析一个实际容灾案例中数据的流动情况。这是一个基于存储设备的自主同步的环境，如图 17.30 所示。

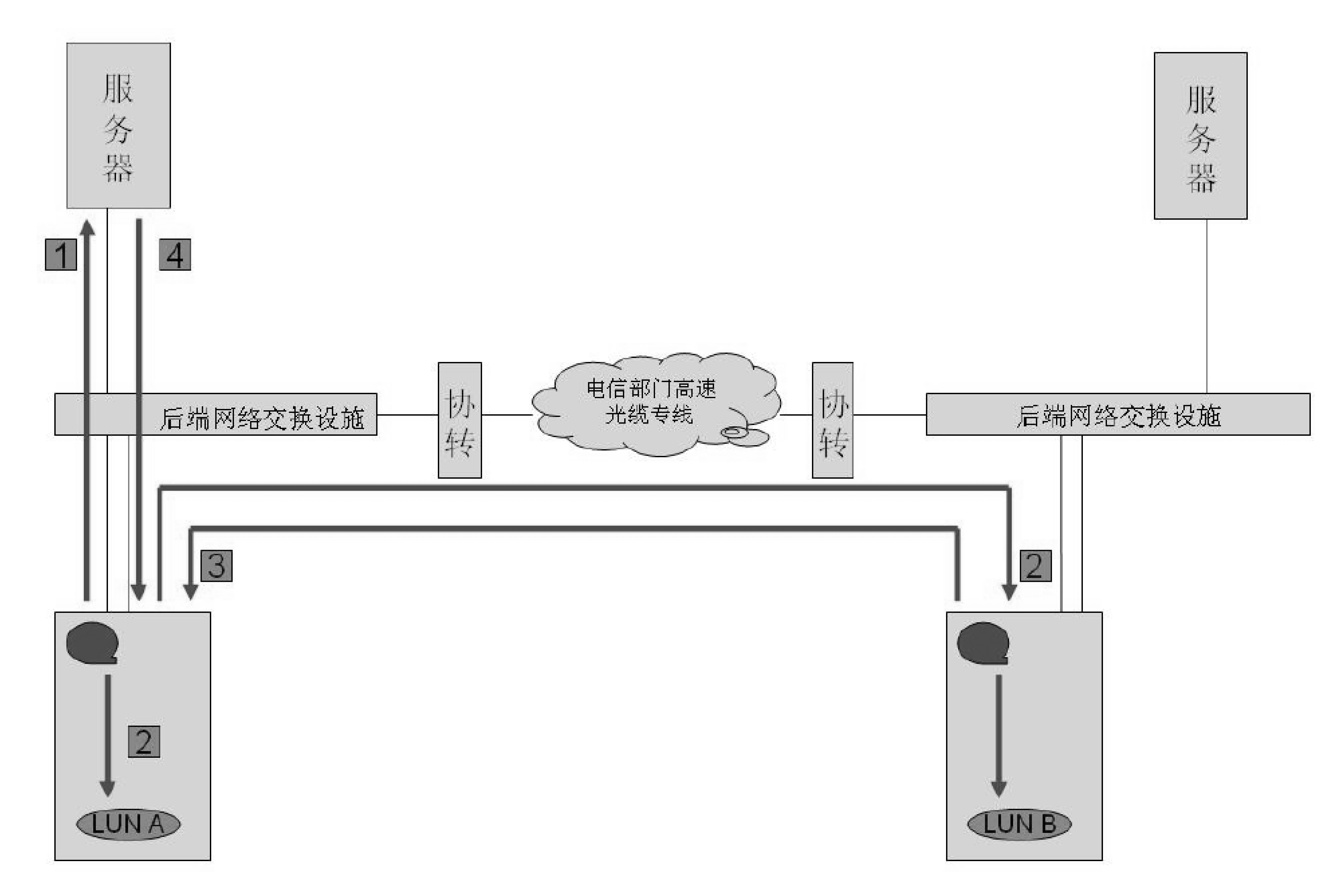


图17.30　同步复制的过程

1】 某时刻，主站点服务器向磁盘阵列发出一个 IO 请求，向某个 LBA 写入数据。待写的数据已经进入了磁盘阵列的缓存中，但是此时磁盘阵列控制器，不会给服务器的 SAN 网络适配器驱动程序发送写入成功的应答，所以发起这个 IO 的应用程序也不会得到写入成功的应答。

2】 主站点磁盘阵列将变化的数据，从缓存中写入 LUN A 中(根据控制器策略，写入一般会有延迟)。与此同时，主站点的数据同步引擎获知到了这个变化，立即将变化的数据块从缓存中直接通过 SAN 交换机发往备份站点的磁盘阵列上的缓存中。

3】 备份站点磁盘阵列上运行的数据同步引擎接收端成功的接收到数据块之后，会在底层 FC 协议隐式的发送一个 ACK 应答，或者通过上层显示的发送给主站点一个应答。

4】 主站点接收到这个应答之后，立即向服务器发送一个 FC 协议的隐式 ACK 应答，这样，服务器上的 FC HBA 驱动程序便会探测到发送成功，从而一层层向操作系统的更上层发送成功信号。最终应用程序会得到这个成功的信号。

如果按照上述方式进行，即如果备份站点的磁盘阵列由于某种原因迟迟未收到数据，则不会发送应答信号，那么主站点的磁盘阵列控制器也就不会给服务器发送写入成功的信号，这样服务器上的应用程序就会处于等待状态，造成应用程序等待，从而连接应用程序的客户端也得不到响应。如果应用程序使用的是同步 IO，则其相关的进程或者线程就会被挂起。这种现象也叫做 IO Wait，IO 等待，意思就是向存储设备发起一个 IO 而迟迟接收不到写入成功的应答信号。如果连接两个站点之间的网络链路出现拥塞、故障，便会发生 IO Wait。

上述的数据复制方式，就叫做同步复制，因为主站点必须等待备份站点的成功信号，两边保持严格的同步，步调一致，一荣俱荣，一损俱损。

17.3.2 异步复制例解

再来看另外一种实现方式，如图 17.31 所示。

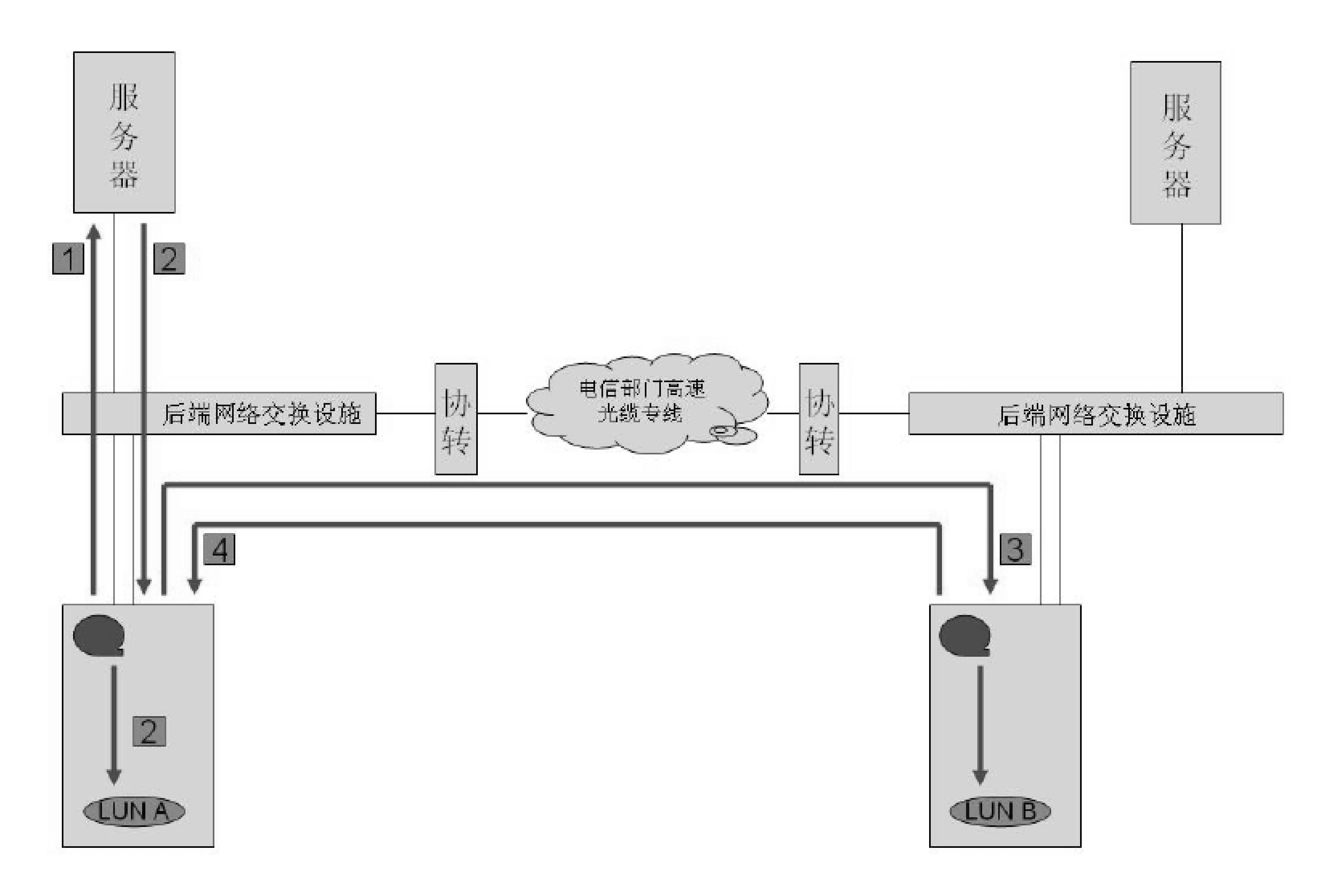


图17.31 异步复制的过程

1】 某时刻，主站点服务器向磁盘阵列发出一个 IO 请求，向某个 LBA 写入数据。待写的数据已经进入了磁盘阵列的缓存中，但是此时磁盘阵列控制器不会给服务器的 SAN 网络适配器驱动程序发送写入成功的应答，所以发起这个 IO 的应用程序也不会得到写入成功的应答。

2】 主站点磁盘阵列控制器根据策略，如果设置为 Write Back 模式，则在第一步之后立即向服务器发送 FC 协议的底层应答。如果设置为 Write Through 模式，则先将数据写入 LUN A，然后再向服务器应答。

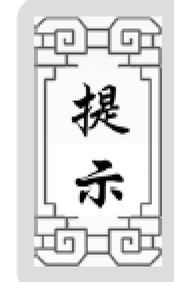
3】 主站点磁盘阵列将这份数据通过 SAN 网络发送给备份站点的磁盘阵列缓存中。

4】 备份站点磁盘阵列成功接收后，返回成功信号。

如果按照上述方式进行，即，主站点磁盘阵列只要接收到服务器写入的数据，就立即向服务器返回成功信号，这样应用程序不需要等待，数据同步动作不会影响应用程序的响应时间。向服务器发送变化的数据，可以在稍后进行，而不必严格同步。这种数据复制的方式就叫做异步复制。也就是说两边步调无须一致，保证重要的事情先完成，不重要的稍后再说。一旦遇到网络连接阻塞或者中断，只要服务器还能访问本地的磁盘阵列，那么应用就不会受丝毫影响，本地磁盘阵列会记录自从网络断开之后，本地卷上所有发生变化的数据块的位置，待网络恢复之后，本地磁盘阵列会根据这些记录，将发生变化的数据块继续复制到远程备份磁盘阵列。

有得必有失。异步复制保证了服务器应用程序的响应速度，然而付出了代价，这个代价就是牺牲了主站点和备份站点数据的严格一致。主站点的数据和备份站点的数据会有一个间隙(GAP)，也就是未被成功复制到远端而积压在本地的数据。此时如果一旦主站点故障发生，这部分数据将永久丢失。而同步复制方式下，没有间隙，如果主站点故障发生了，

备份站点的数据就是主站点发生故障那个时刻的严格数据镜像，不会有数据丢失。同样，同步复制的代价，就是牺牲了服务器的应用程序响应时间。



在实际容灾系统设计的时候，一定要考虑这一点，要明白用户是愿意牺牲数据安全性来换取高响应时间，还是愿意牺牲响应时间来换取数据的安全性。

目前很多设备厂商都有折中的解决方案，比如在网络正常的情况下，实现同步复制，一旦检测到网络连接超时，则转为异步复制，待网络正常后，再转为同步复制。

17.4　生产者的容灾——服务器应用程序的容灾

IT 系统的生产者，也就是各种服务器上运行的应用程序。毫无疑问，主站点发生故障，必须要在备站点重新运行这些应用程序。我们是否可以在备份站点预备应用程序的安装文件，发生故障后，在备份站点服务器上安装配置这些应用程序呢？

这么做虽然可行，但是一些较为复杂的应用程序，安装和配置要花费大量的时间，比如 SAP 企业 REP 系统的安装，可能需要一天的时间，再加上不可预料的因素，耗时可能更长。如果没有预先安装配置好这些应用程序，未雨绸缪，则事故发生的时候，企业就需要忍受停机所带来的损失了。

17.4.1　生产者容灾概述

我们必须将应用程序在备份站点预先安装并且配置好，但是不能让它们处于工作状态，应当时刻保证同一时刻只有一个站点的生产者在生产，因为 IT 系统生产出来的产品是具有一致性的数据，而且数据是有时效的，具有上下文联系的。IT 生产是一个连续的数据处理过程，一旦中途产生数据不一致性，就需要恢复数据到某个一致的时刻，然后从这个时刻继续生产。而不像实物生产那样，产品是一件件的物品，都具有相同的属性。所以保证同一时刻，整个 IT 系统只有一个站点的生产者处理同一份数据，这一点非常重要。

然而，既要求两个站点同一时刻只能有一个站点的生产者处理一份数据，又要求当生产站点发生事故的时候，备份站点的生产者立即启动，接着处理备份站点经过主站点数据同步过来的数据，做到这一点，就需要让备份站点的应用程序感知到主站点应用程序的状态，一旦检测到主站点应用程序故障，则备份站点应用程序立即启动，开始生产。

第 16 章曾经说过高可用性群集，而在容灾技术领域中，群集的范围扩大到了很远的范围，备份站点与主站点可能不在同一机房中而在相隔很远的两座建筑物里，甚至两个城市中。这样，备份应用程序就要跨越很远的距离与主应用程序通信来交换状态。由于应用程序运行状态数据，相对于其处理的数据来说，数据量是很小的，所以即使是跨越广域网通信，也不必担心延迟太大。如果是通过广域网连接两个站点的前端网络，则最好使用专线连接；如果是基于 Internet 的 VPN 连接，虽然可以获得高带宽/价格比，但是延迟无法保证最小，除非购买电信部门提供的 QOS 服务。

类似 HACMP、MSCS 这种 HA 软件，都是使用共享存储的方式来作用的，即 HA 系统中

的所有节点，共享同一份物理存储，不管某时刻由谁来操作处理这些数据，最终的数据只有一份，而且是一致的、具有上下文逻辑关系的。而远程异地容灾系统中，数据在主站点和备份站点各有一份，而且必须保证两边数据的同步。

生产的时候，必须以一边数据为准，另一边与之同步，绝对不能发生两边同时进行生产的情况，除非两边生产者处理的是两份逻辑上无任何关联的数据。所以，远程容灾系统所要关注的有两个重要因素，即生产者和生产资料。只要生产资料在主站和备站完全同步，那么就可以逻辑上认为，数据只有一份，备站的数据是主站的镜像，平时虚无缥缈不可用，但是一旦主站发生故障，备站的镜像立即成为实实在在可用的数据，同时，生产者在备站启动生产，处理数据。这就是异地容灾。

然而，传统的基于共享存储模式的 HA 软件，不适用于异地容灾系统，因为共享存储模式的 HA 软件，是基于资源切换为基础的，它把各个组件都看成是资源，比如应用程序、IP 地址、主机名、应用所要访问的存储卷等，发生故障时，备份机 HA 软件检测到对方的故障，然后强行将这些资源迁移到本地，比如，在备份机修改相应网卡的 IP 地址，并发出 ARP 广播来刷新所有本广域内的客户端以及本地网关设备所保存的 ARP 映射记录，以让所有网络上的终端获知此 IP 对应的新 MAC 地址，修改主机名映射文件(Host 文件)，挂载共享存储设备上的卷，最后启动备份应用系统。

应用系统可以访问已经强行挂载的共享卷而存取数据，客户端可以继续使用原来的 IP 地址来访问服务器上运行的应用程序，因为这个 IP 已经由故障的计算机转移到了备份计算机，这样，生产就可以继续进行了。要保证生产者在切换之后生产可以继续，则必须先保证生产者所依赖的所有条件已经切换成功，这些条件包括 IP 地址(非必须)和卷等。

1. 本地容灾系统中的两种存储模式

本地 HA 系统中，多个节点如果共同拥有同一个或者同几个卷，但是同一时刻只有活动节点才挂载该卷进行 IO 读写，这种模式就叫做**共享存储模式**，即 HA 系统中的每个节点都拥有同一份存储卷，只不过不活动的节点不对其进行挂载并 IO。

如果 HA 系统中每个节点都有自己独占的存储卷，这些卷除了拥有者可以读写之外，任何情况下，其他节点都不能读写，数据的共享是通过同步复制技术同步到所有节点上的存储卷中的。这种方式就叫做 **Share-Nothing 模式**，即 HA 系统中的所有节点之间不共享任何东西，所有元素都是独享的，甚至网络地址都是各用各的。数据存在多份，每个节点一份，节点之间通过同步复制技术来同步数据，某节点发生故障之后，这个节点对应的备份节点直接启动应用程序，由于之前数据已经在所有节点上同步，所以此时数据是完整一致的。由于 Share-Nothing 模式下，不存在任何的“接管”，所以此时客户端需要感知到服务端群集的这种切换动作，并通过客户端手动或者自动切换配置以便连接新服务器。表 17.3 对比了 HA 的两种存储模式

表 17.3 HA 群集中两种存储模式的对比

	共享存储	Share-Nothing
数据本身是否容灾	否	是
软硬件成本	高	低

续表

	共享存储	Share-Nothing
前端网络资源耗费	低	高
管理难度	高	低
维护数据是否需要停机	需要	不需要
实现复杂程度	高	低
是否需要第三方软件	是	否
故障因素数量	3 个	2 个

- 数据本身是否容灾
  共享存储模式下，容灾系统的各个节点共享同一份数据。如果这份数据发生损坏，则必须用备份镜像加以还原，而且需要承受停机带来的损失。而 Share-Nothing 模式下，系统中每个节点都有自己的数据拷贝，如果其中一份数据被破坏，系统可以切换到另外的节点，不影响应用，不需要停机，被损坏的数据可以在任何时候加以还原修复，并且修复后的节点可以再次加入容灾系统。
- 软硬件成本
  共享存储模式下，由于各个节点需要共享一份存储数据，所以需要外接的磁盘阵列系统，而且为了保证数据访问速度，外接存储系统必须自身实现 RAID 机制，主机上也需要安装连接盘阵的适配器。这样就增加了整个系统的成本。Share-Nothing 模式下，各个节点自身保存各自的数据，而不必使用外接存储系统。另外，共享存储模式还需要额外的 HA 软件及额外的成本，而 Share-Nothing 模式不需要。
- 前端网络资源耗费
  共享存储模式下，各个节点之间交互信息一般通过以太网络，而存储数据通过后端存储网络。由于各个节点在前端网络上只传输控制数据，所以对前端以太网络资源的耗费相对较低。而 Share-Nothing 模式下，由于各个节点之间的数据同步完全通过前端网络，所以对前端网络资源耗费相对较高，适合局域网环境。
- 管理难度
  共享存储模式下，不但需要管理节点间的交互配置，还需要管理外部存储系统，增加了管理难度。Share-Nothing 模式下，只需要管理各个节点见的交互配置即可。
- 是否需要停机
  共享存储模式下，由于需要将数据从单机环境转移到共享存储环境供其他节点使用，往往需要停机来保证数据的一致性。而 Share-Nothing 模式下，数据同步是动态的，不需要停机。
- 实现复杂程度
  首先，共享存储模式下，有三种基本元素：节点、节点间交互、共享数据，而 Share-Nothing 模式下，只有两种元素：节点、节点间交互。其次，如果使用共享存储模式做容灾，需要将数据移动到共享存储上，增加额外的工作量、时间和不可控因素。

- 是否需要第三方软件

  共享存储模式下，备份节点需要通过第三方软件来监控主节点的状态，在发生故障的时候主动接管资源，比如各种操作系统提供的 HA 软件(HACMP、MSCS、SUN Cluster 等)。Share-Nothing 模式下不需要任何第三方软件参与。
- 故障因素数量

  共享存储模式下，如果出现容灾系统本身的功能故障，需要在操作系统、应用程序、HA 软件三个方面排查故障。Share-Nothing 模式下，只需要在操作系统、应用程序二者之间排查故障。

2. 异地容灾系统中的 IP 切换

在异地容灾系统中，主服务器和备份服务器不太可能在一个广播域中，一般都是通过网关设备来转发之间通信的 IP 包，所以不可能用所谓资源切换的方式来切换 IP 地址。如果想对客户端透明，即客户端可以无须感知故障的发生，继续使用原来的 IP 地址来连接备份服务器，那么就需要在网络路由设备上做文章了，动态修改路由器上的路由表，将 IP 包路由到备份站点而不是主站点。如果客户利用域名来访问服务器，那么也可以直接在 DNS 设备上修改 IP 指向记录来完成这个功能。

最方便而且普遍的做法是：让所有客户机利用主机名来连接服务器，这样，主站点故障后，通知所有客户端修改它们的 host 文件即可将原来的主机名映射到新的 IP 地址而不用重启计算机。这方面，异地容灾系统中的 HA 软件几乎发挥不了作用。

3. 异地容灾系统中的卷切换

异地容灾系统中在主站点和备站点各有卷，两个卷之间可以通过前端网络同步，或者通过后端网络同步。主站点后，备份服务器上的 HA 软件检测到主服务器通信失败，便会感知故障发生，然后通过某种方式，断开主卷和备份卷的同步关系(如果不断开，则卷会被锁定而不可访问)，如果同步引擎是运行在存储设备上的，那么除非 HA 软件可以操控运行在存储设备上的同步引擎，否则必须由系统管理员手动利用存储设备的配置工具来断开同步关系。同步关系断开后，本地的卷才能被访问，这样，HA 软件才能在备份机上调用操作系统的相关功能来挂载这个卷。

如果同步引擎本身就是由运行在主机和备份机上的 HA 软件提供的，那么就可以实现在检测到通信失败之后，由 HA 软件本身来自动断开同步关系，然后在备份机上挂载对应的卷。

4. 异地容灾系统中的应用切换

应用，也就是生产者的切换，是所有 HA 容灾系统中的在故障发生后所执行的最后一步动作。与共享存储模式的 HA 容灾相同，异地容灾中的应用切换，也是由备份机的 HA 软件来执行脚本，或者通过其他功能调用相关应用的接口来启动备份机的应用。

比如，对于 DB2 数据库来说，启动数据库实例所使用的命令为：db2start，HA 软件只要检测到主站点故障，只要在备份机的 db2cmd 命令行方式下执行这条命令，便可使备份机的 DB2 数据库实例启动起来。应用的启动必须在所有资源成功切换到备份机后发生，因为应用启动的时候必然会读取卷上的一些数据，如果卷还没有被挂载，应用启动的时候就会报错，比如：找不到数据文件。

5. Veritas Cluster Server 软件介绍

VCS(Veritas Cluster Server)可以基于 VVR 的配合，而实现异地容灾系统。在一个 CLUSTER 环境中，如果一台服务器运行多个应用，只有一个应用出现故障时，那么 VCS 可以只将该应用切换到预先定义的服务器上，另一个应用仍然在原来的服务器上继续运行。

VCS 将其监视的应用当作一组资源来管理，这一组资源定义为资源组(RG)。例如 Web-Server，要保证这个应用正常运行，VCS 将监视存放数据的磁盘组，该磁盘组上的文件系统、网卡、IP 地址及 Web 服务进程。既然 VCS 是基于应用的高可用软件，一台服务器上运行的多个应用可以切换到不同的服务器上。

例如，图 17.32 所示的服务器 A 运行着 IIS 网页访问服务和 DB2 数据库服务，服务器 B 运行着 FTP 服务和邮件转发服务，服务器 C 运行着 NFS 服务和 SMB 网络文件系统服务。当服务器 A 出现故障时，资源组 RG-Web 切换到服务器 B 上，资源组 RG-DB2 切换到服务器 C 上。当然条件是它们都能存取对应的应用数据。系统管理员制定合适的故障条件，例如现场完全瘫痪 10 分钟或某个应用停止运行半小时。当这种情况发生时，可以设定有 GCM(Global Cluster Manager)自动切换应用，或向系统管理员报警，得到确认后，再切换应用。无论应用切换是自动还是需要确认，两个场地之间应用的启动过程均无须人工干预。

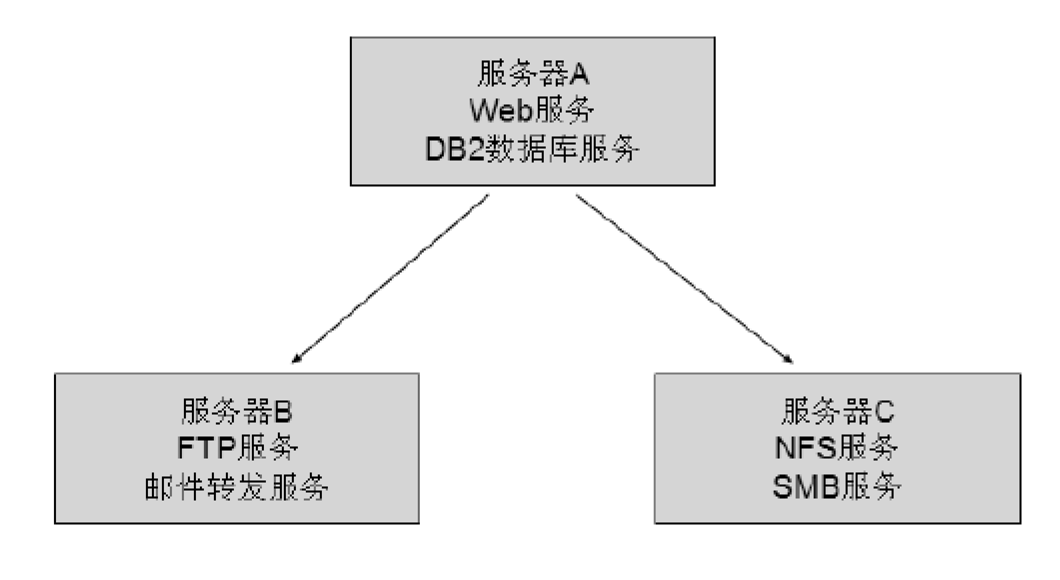


图17.32 多 Active 集群

17.4.2 案例一：基于 Symantec 公司的应用容灾产品 VCS

图 17.33 所示为两台 DB2 数据库服务器，下面要将其配置为一个 HA 双机热备系统，主机硬件或者应用程序故障之后，由 VCS 自动检测故障，并在备份机上重新启动各种环境以及应用程序。

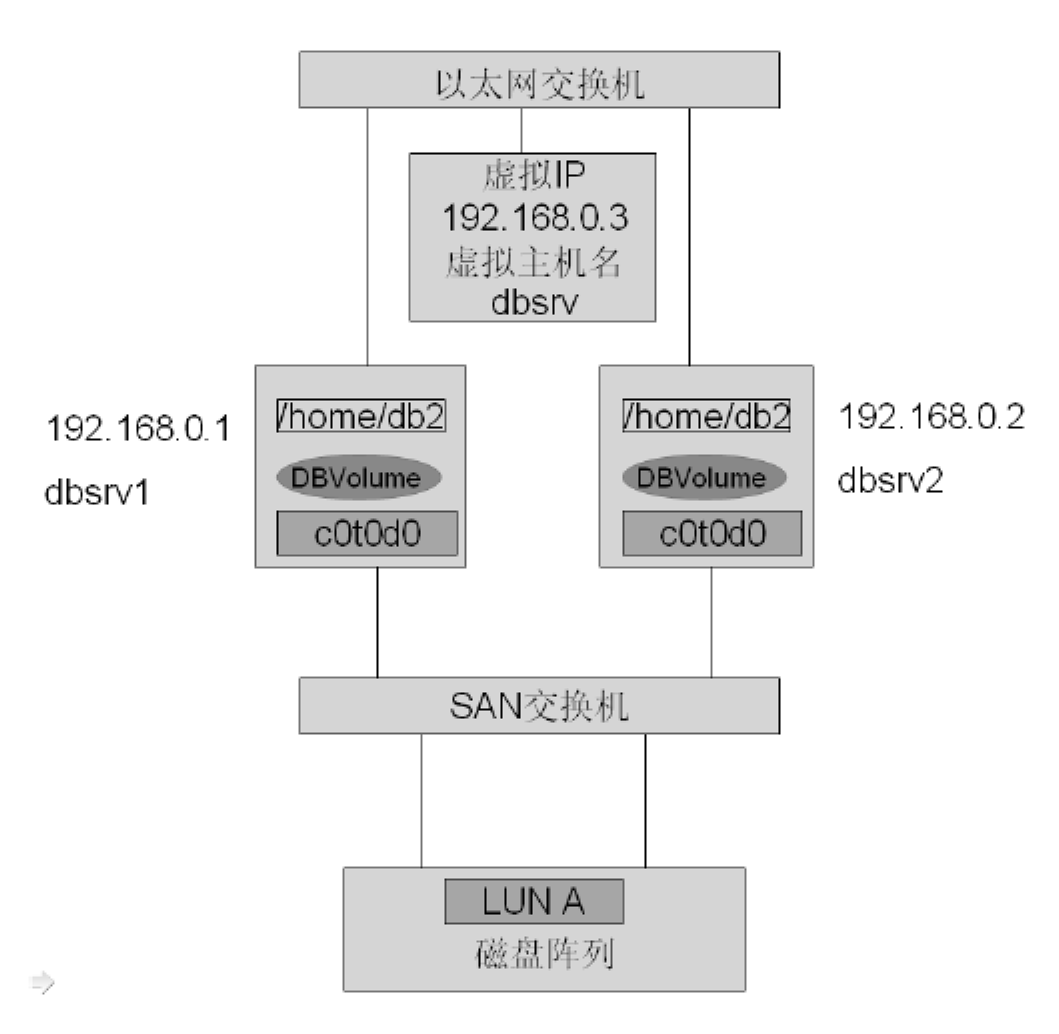


图17.33 两台服务器的双机系统

主服务器名称为 dbsvr1，IP 地址为 192.168.0.1，备份服务器名称为 dbsvr2，IP 地址为 192.168.0.2。

两台计算机操作系统都是 Solaris 9，利用 Symantec 的 Storage foundation(包含了 VxVM 和 VxFS)作为卷和文件系统管理工具。

在两个系统中分别安装了 DB2 数据库程序，而数据库文件存放在共享磁盘阵列上面，共享卷由 VxVM 对底层磁盘进行虚拟化而生成，VxVM 先将操作系统底层磁盘(盘阵上的 LUN)组成磁盘组，然后在这个组中再划分卷，这就和 RAID 卡的做法类似，只不过 VxVM 作用在主机操作系统层，而 RAID 卡作用在硬件层。

本例所生成的共享磁盘组命名为 DBDG，只包含一个物理磁盘，设备名为 c0t0d0，然后划分一个卷，卷名 DBVolume，用 VxFS 格式对这个卷进行格式化，再将格式化好的卷挂载到虚拟目录中，本例将其挂载于/home/db2 下面。

对卷的划分和格式化仅需在一台计算机上配置即可，配置完后，只需要将这个卷进行导出操作，另一台计算机就可以导入并识别出这个卷的格式，再直接挂载到虚拟目录。

为了对客户端透明，我们用一个虚拟主机名和虚拟 IP 作为访问 DB2 数据库服务的地址，虚拟主机名为 dbsvr，虚拟 IP 为 192.168.0.3。

虚拟 IP 不是一个神秘的东西，我们知道一块以太网卡可以有多个 IP 与之对应，如果把 192.168.0.3 这个 IP 绑定到 dbsrv1 主机的网卡上，那么 dbsrv1 主机就同时拥有两个 IP：192.168.0.1 和 192.168.0.3，这样，客户端用 ARP 协议请求 192.168.0.3 这个 IP 地址对应的 MAC 地址时，dbsrv1 这台主机便会应答，客户端知道 192.168.0.3 这个地址的 MAC 地址(dbsrv1 主机网卡的 MAC 地址)，就可以建立与 dbsrv1 主机的通信。

一旦 dbsrv1 主机发生故障，那么 dbsrv2 主机上的 VCS 软件就会将 192.168.0.3 这个 IP 地址设置到 dbsrv2 主机的网卡上，并发出 Free ARP 广播，将新的 IP 与 MAC 地址的对应关系通告到网络上的其他终端，客户机再次连接的时候，就会建立和 dbsrv2 主机的通信，而客户端对这个 IP 的拥有者是 dbsrv1 还是 dbsrv2 丝毫没有察觉，也没有必要察觉到。这个切换 IP 的动作，也是 VCS 将虚拟 IP 作为一个资源来切换的过程。



Storage Foundation 的安装过程这里就不做描述了，本例假设在两台计算机上都已经成功的安装 Storage Foundation 组件了。

相关配置配置过程

1】 在 dbsrv1 主机上创建供 DB2 数据文件使用的共享存储，及文件系统。

```
# vxdg init DBDG c0t0d0 \\创建磁盘组 DBDG，使用 c0t0d0 这个硬盘
# vxassist -g DBDG make DBVolume 5g \\在磁盘组上创建 5GB 大小的卷 DBVolume
# mkfs -F vxfs -o largefiles /dev/vx/rdsk/DBDG/DBVolume \\将卷 DBVolume
格式化为 VxFS 文件系统
# mkdir /home/db2  \\创建挂载点，将用于 DBVolume 卷的挂载
# mount -F vxfs /dev/vx/dsk/DBDG/DBVolume /home/db2  \\将格式化好的卷
DBVolume 挂载于/home/db2 下，这样就可以通过 CD /home/db2 进入这个目录从而对这个
卷的内容进行访问了。
```

2】 使两个系统可以通过 RSH 方式互相访问，在 dbsrv1 上面做如下操作。

```
# echo "dbsrv2  192.168.0.2" >> /etc/hosts \\将对方加入自己的主机列表
# echo "dbsrv2 db2inst1" >> $HOME/.rhosts  \\使得对方主机可以通过 RSH 以
db2inst1 的身份登录本机。Db2inst1 是 DB2 数据库所必须的用户
# echo "dbsrv 192.168.0.3" >> /etc/hosts  \\将虚拟主机加入自己的主机列表
```

3】 在 dbsrv2 上面做类型的操作，将 dbsrv2 改为 dbsrv1，IP 也做相应的改变，虚拟主机 IP 和主机名不变。

4】 在两台计算机上分别执行下列命令，创建相同的用户组。

```
# groupadd -g 999 db2iadm1  \\创建 DB2 实例管理组;
# groupadd -g 998 db2fadm1  \\创建 DB2 fencing 管理组;
# groupadd -g 997 db2asgrp  \\创建 DB2 数据库管理组;
# useradd -g db2iadm1 -u 1005 -d /home/db2 -m db2inst1  \\创建 DB2 实例
管理用户
# useradd -g db2fadm1 -u 1006 -d /home/db2fenc1 -m db2fenc1  \\创建 DB2
fencing 管理用户
# useradd -g db2asgrp -u 1007 -d /home/db2as -m db2as  \\创建 DB2 数据库
管理员账户
```



上述用户组或者用户的 ID 可以是尚未被使用的任意数字，但一定要保证两台计算机上面的用户 ID 是一致的，否则数据库切换的操作会失败；数据库实例管理员的账户目录要存放在共享盘上面，也就是/home/db2 目录。

5】 在两台计算机上面分别安装 DB2 数据库程序。

用 install 程序来安装 DB2，然后手动创建实例和数据库。因为实例目录需要放到共享卷上，也就是/home/db2 目录。

6】 安装完 DB2 程序后，分别在两台计算机安装 DB2 的许可证。

```
# /opt/IBM/db2/V8.1/adm/db2licm -a db2ese.lic
```

7】 在 dbsrv1 上面创建实例(存放到共享盘)。

```
# cd /usr/opt/db2_08_01/instance
# ./db2icrt -u db2fenc1 db2inst1  \\创建一个名为 db2inst1 的实例，DB2 会将实
例目录存放到同名的用户名目录下，也就是 dbinst1 用户的主目录：home/DB2 目录下，从
而将实例目录放到了共享卷上。
```

8】 修改 DB2 节点文件/home/db2/sqllib/db2nodes.cfg，将原来的 db2srv1 主机名修改为 dbsrv 这个虚拟主机名。

```
0 dbsrv 0
```

9】 创建数据库 testdb。

```
# su - db2inst1 \\切换数据库实例管理用户;
# db2start     \\启动数据库;
# db2 create database testdb  \\创建新的数据库 tdstdb，由于当前用户是
db2inst1，所以 testdb 数据库被创建在/home/db2 目录下，也就是共享卷上;
# db2 terminate  断开与 DB2 服务后端处理进程的连接;
# db2stop   \\停止数据库;
```

10】 将共享盘从 dbsrv1 卸载下来(在 dbsrv1 执行)。

```
# umount /home/db2  \\卸载文件系统；
# vxvol -g DBDG stopall  \\将 DBDG 的所有卷停止活动；
# vxdg deport DB2DB  \\将磁盘组 DBDG 导出，以便在其他计算机上导入并挂载。
```

11】 将共享盘挂载到 dbsrv2(在 dbsrv2 执行)

```
# vxdg import DBDG  \\将磁盘组 DBDG 导入；
# vxvol -g DBDG startall  \\将 DBDG 的所有卷启动；
# mount -F vxfs /dev/vx/dsk/DBDG/DBVolume /home/DB2  \\挂载文件系统；
```

12】 在 dbsrv2 启动原来在 dbsrv1 创建的数据库 testdb：

```
# su - db2inst1
# db2start
# db2 connect to testdb
```

如果能够连接成功，则数据库双机配置成功。如果数据库服务在某系统上发生故障后，会被 VCS 切换到另外一台计算机并运行。下面配置自动故障检测并切换的功能。

13】 拷贝 DB2 代理配置文件到 VCS 的配置目录。

```
# cp /etc/VRTSvcs/conf/Db2udbTypes.cf  /etc/VRTSvcs/conf/config/
Db2udbTypes.cf
```

14】 打开 VCS 图形工具。

```
# /opt/VRTSvcs/bin/hagui  \\将运行 VCS 图形化配置工具
```

15】 创建服务资源组(service group)，并命名为 db2grp。

16】 依次单击“文件”→“导入”→“确定”按钮，导入 DB2 代理配置文件。

17】 在 db2grp 中创建六个资源。

- 磁盘组：即 DBDG。
- 卷：DBVolume。
- 挂载点：/home/db2。
- 网卡：客户端所连接的网卡(例如 bge0)。
- IP 地址：选择 192.168.0.3 这个虚拟 IP 地址。
- DB2 agent：这个资源会监控 DB2 程序在群集中的运行情况。

18】 为这六个资源创建依赖关系(右击资源，选择 link)。

IP 依赖 NIC 网卡的工作正常；卷的存在依赖于磁盘组的状态；文件系统依赖卷；DB2 代理的状态要同时依赖于 IP 地址的存在和文件系统的存在。

19】 右击 db2grp 服务组，选择 online，让 db2 在 dbsrv1 上线。

20】 右击 db2 服务组，选择 switch to，让 db2 切换到 dbsrv2。

21】 如果切换正常，则 VCS 配置成功。

17.4.3 案例二：基于 Symantec 公司的应用容灾产品 VCS

图 17.34 是两个站点容灾系统最基本的结构图。主站点是基于三个节点 Cluster 的多个应用，容灾站点同样也配置成一个三节点的 Cluster 系统，它们配了同等容量的存储，并具

有数据容错功能。

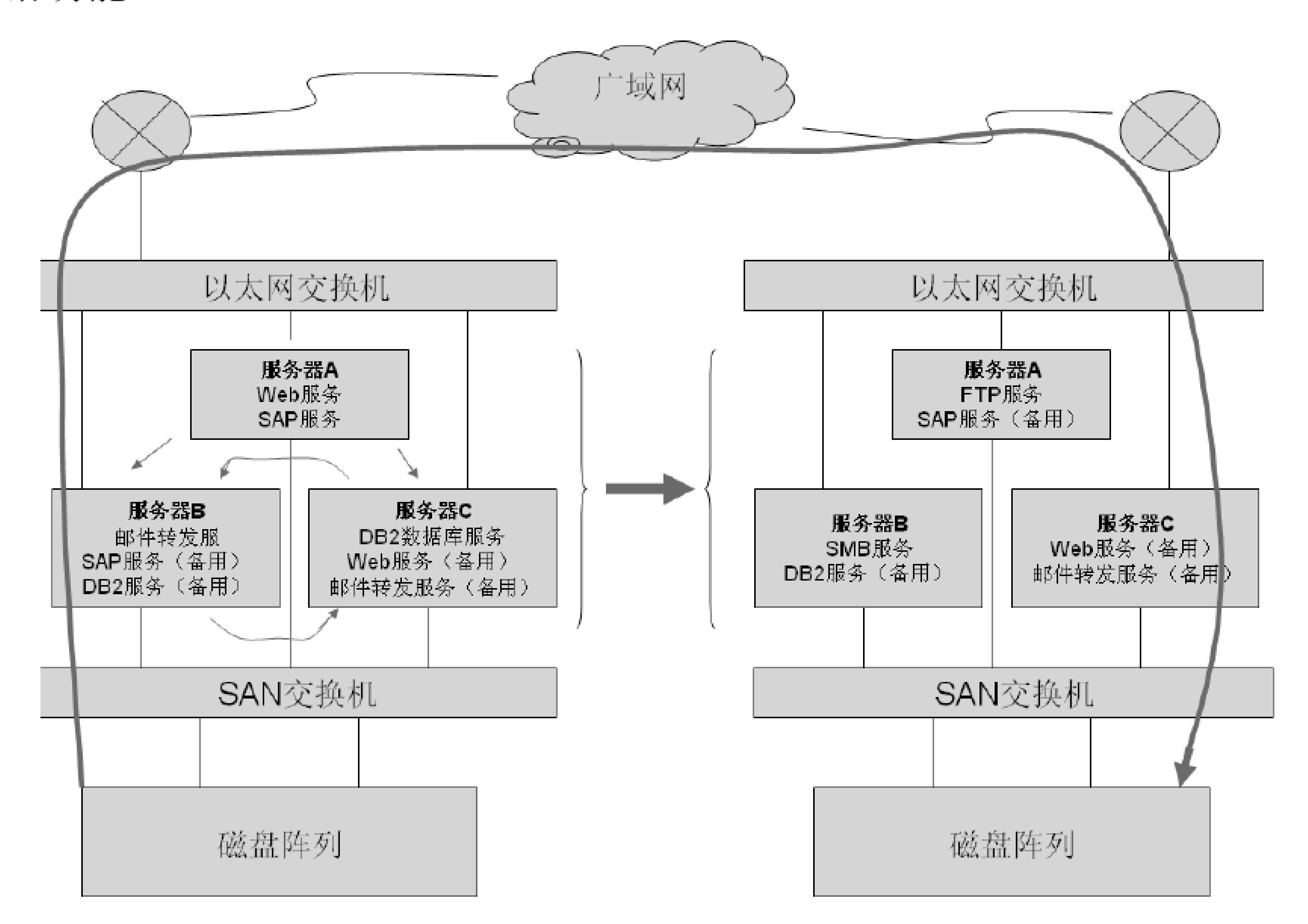


图17.34　两个站点的互备群集

主站点运行的是 Web 服务、DB2 服务，SAP 企业 ERP 服务以及邮件转发服务的关键业务，完全置于容灾系统控制之下，可以看到：

- 主站点服务器 A 上安装了 Web 服务程序和 SAP 应用程序，而且二者皆在运行状态。
- 服务器 B 上安装了邮件转发处理程序，以及 SAP 应用程序和 DB2 数据库程序，但是服务器 B 上的 SAP 和 DB2 程序平时都处于停止状态，只有邮件转发程序在运行。
- 服务器 C 上安装了 DB2 数据库程序，并处于运行状态，另外还安装有 Web 服务程序和邮件转发程序，但是平时处于停止状态。

备站点的业务是 FTP 服务、SMB 文件共享的一般业务，不做容灾。备份站点的服务器 A 上运行的是 FTP 服务程序，同时安装有 SAP 应用程序，但是 SAP 应用程序处于停止状态；服务器 B 上运行着 SMB 文件共享服务，同时安装有 DB2 数据库服务程序，但是 DB2 数据库服务程序平时处于停止状态；服务器 C 上安装有 Web 服务程序和邮件转发程序，并且都处于停止状态。

如图 17.34 所示，图中最长的箭头指示了两个站点数据同步的路径，即从主站点盘阵(或者主站点服务器的内存)，经过前端以太网交换机，传送到网关设备，然后经过广域网到达备份站点的网关设备，再通过前端以太网交换机传送到备份站点服务器内存，最后从内存写入后端磁盘阵列。

主站点服务器之间的箭头，表示一旦某个服务器，或者服务器上的某个应用发生故障之后，资源组的切换走向。

从图 17.34 中可以看到服务器 B 和服务器 C 形成了一个互备的系统，即服务器 B 是邮件转发程序的主节点，是 DB2 服务的备用节点；而服务器 C 是 DB2 服务程序的主节点，是邮件转发程序的备用节点。



因为主站点的三台服务器之间形成了比较复杂的互备关系，所以三台服务器必须能识别到其他两台服务器上挂载的卷，但是备用节点不应当挂载这些卷，仅当对方应用或者整个服务器故障的时候，才能在备份节点上挂载这些卷。

图中央的粗箭头，表示一旦主站点发生了大故障，诸如整个机房被损毁等，那么所有主站点的应用，全部切换到备份站点，并且备站点的节点挂载备用磁盘阵列上的所有卷。Veritas 的 Storage Foundation 组件应当安装到图上的所有服务器中，VVM 模块用于管理所有存储卷，VVR 模块用于同步所有卷的数据，VCS 模块用于检测故障并且切换应用。整个 HA 系统的工作过程如下。

1】 主站点应用的运行过程中，所修改的数据通过所有主站点服务器主机都安装的 VVR 软件实时地复制到备站点。

2】 假设某时刻主站点服务器上的 SAP 应用发生故障，比如相关服务无法启动，则 VCS 模块检测到这个故障之后，发现相关资源组有两个备用节点可切换：主站点的服务器 B 和备站点的服务器 A，所以它首先检测主站点的服务器 B 是否可用，如果可用，则发送一些信息通告服务器 B 上的 VCS 模块，准备切换 SAP 应用到服务器 B，服务器 B 确认后，VCS 在服务器 A 上首先卸载 SAP 程序所存储的对应卷，然后通告服务器 B 卸载成功，服务器 B 再挂载这些卷，并且接管 SAP 服务所利用的 IP 地址，之后启动 SAP 服务。客户端只需要重新连接一下便可。

3】 某时刻，主站点机房供电系统故障，经过相关人员的检查，恢复供电大概需要 5 小时。而 UPS 系统在工作两小时之后因电量不足而停止，企业 CIO 果断决定，在 UPS 电量耗尽之前，将主站点所有系统手动停机以免因为突然断电对硬件和软件带来的损害。此时备用站点的 VCS 软件检测到了这个故障，立即在所有服务器上挂载已经经过数据同步的卷，然后启动所有备份应用系统。所有生产均恢复运行，客户端经过修改 host 文件或者修改所连接的 IP 地址，恢复了与服务器的连接，所有生产继续运行。

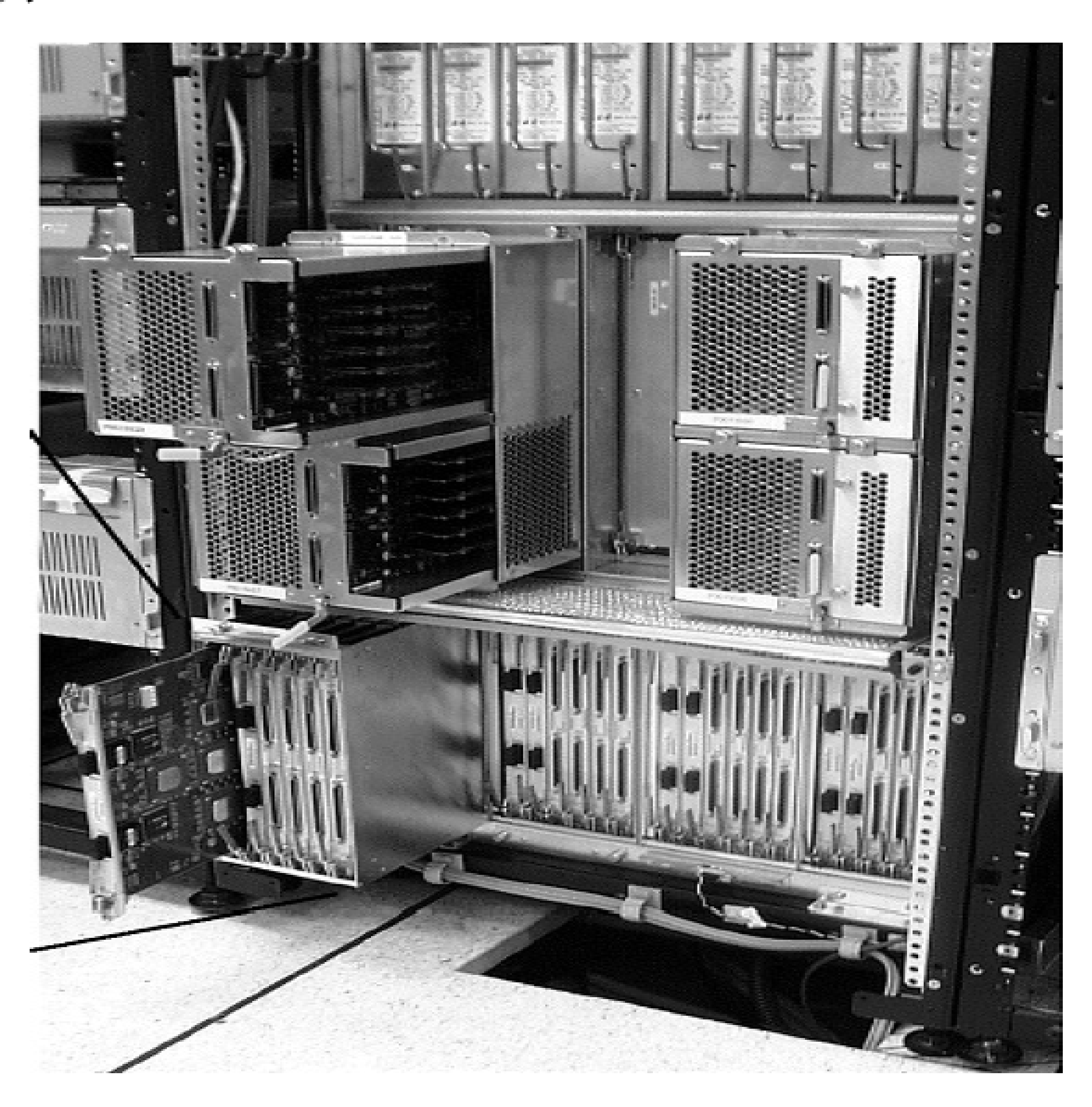
4】 5 小时之后，主站点机房供电恢复，UPS 系统充电。企业 CIO 决定，在恢复供电 1 小时之后(确保主站点供电恢复正常，以避免不必要的动荡)，切回所有应用到主站点。主站点所有系统开机，VCS 软件会检测到当前的应用已经全部运行在备份站点，所以不会在主站点服务器上挂载卷并启动应用。与此同时备用站点的 VVR 软件重新建立了与主站点 VVR 软件的通信，并互相交互数据，备份站点的 VVR 检测到了主站点相关卷上的数据是落后的，因为备用站点在主站点故障期间，已经生产运行了五小时，此间数据已经有所变化。VVR 立即将变化的数据复制到主站点的相关卷。重新同步后，备用站点的 VCS 停止应用、卸载卷，主站点的 VCS 挂载卷，启动应用，所有状态恢复如初，客户端重新连接即可连接到主站点的服务器上。

本次故障造成的停机时间很短，没有对生产造成太大影响，同时也很好的考验了这个企业的异地容灾系统的功能。如果没有容灾系统，这个企业就要忍受长时间停机带来的损失。

读 书 笔 记

附录　五百年后——系统架构将走向何方

附图 1 为一台大型机设备的一角，底部的 24 个 IO 插板不但占据了庞大的空间，也耗费了太多的电力和人力。



附图1 大型主机一角

目前，Intel 已经将 CPU 与 GPU 的整合计划提上日程，将在下一代 CPU 中整合入显示芯片，整合的 GPU 性能可满足一般的 3D 应用。不但如此，内存控制器也将从北桥迁移到 CPU 芯片中。

SUN 公司正在到处宣传他们的 Server on chip(芯片上的服务器)理念。要将 10Gb/s 速率的以太网适配器集成到 CPU 芯片中。将多个 CPU 核心集成到一个芯片里。

摩尔定律依然在实现它的预言，电路集成度越来越高，外部单个系统架构也将越来越简单。而网络技术的发展必将进入一个超级网络化世界。固态存储将很高的容量集成在一个很小的空间，此时还需要 SAN 么？那时候的 SAN 将是另一种含义，即 System Area Network。即联网的是系统总线，而不是后端存储。

分还是合？目前的存储界，有人在做整合，有人在做分离。分分合合，真的不知道谁能最终主宰存储界。SAN 的出现，本来是为了集中存储，服务器只管向 SAN 上的盘阵系统要数据或者写数据，而不必在自己机箱内部插满硬盘。 但是 SUN 公司将 24TB 的容量集成在一个标准 4U 机架服务器中，让服务器不用再连接 SAN，不需要购买 FC 适配卡，不需要 FC 交换机，不需要连接光纤。前者为“分”而来，后者为“合”而去。目前 IT 系统所处的阶段正是一个混沌阶段，各种迹象均表明，系统正在走向整合收缩阶段。

一些厂家的服务器正体现了这种趋势，单个 4U 高度机柜中可以存储几十 TB 的容量(SATA 硬盘)，如果用体积和耗电更低的固态存储硬盘，其存储密度将会更高。到撰写本章时，单块 SSD 的读速率已经超过了 200MB/s，写速率也超过了 160MB/s，随机 IO 的 IOPS 远远超越机械硬盘。

随着芯片存储技术的发展，也许在一张电路板上就可以达到 100TB 的存储容量，那么这种密度还需要用光缆连接存储设备来获得数据流么？那时候的 SAN 就像并行 SCSI 一样，将被抛弃。没有人愿意连接光缆到外部存储设备。

SUN 公司的 BlackBox 计划，也体现了“整合”的思想，它试图将 IT 系统引向收缩的趋势。SUN 公司的 BlackBox 将企业的 IT 中心放入一个集装箱中，其中包含了电源、机架、网络设施、服务器主机、存储设备、监控设备、散热冷却等所有数据中心机房的部件。如果某个企业打算迁移到另外的地区，则直接将这个集装箱运往目的地即可。

我们可以乐观地认为，随着芯片整合度的飞速提高，将来企业的 IT 中心，将会是一块手掌大小的芯片，可以随身携带。

IT 领域中永恒的话题，终归还是网络技术。甚至在五百年后，网络技术可能成为人类自身赖以发展的基本。探索和交流，乃是人类世界和计算机世界的永恒，而交流必须使用的就是网络技术。网络的例子还有很多，比如计算机总线网络、交通网络、电话网络……

网络技术将走向何方？至少可以肯定的是更高速、更稳定、传输距离更远。可以推测的是：无线传输最终会替代有线传输。从 2005 年开始，笔记本电脑已经普遍安装了 802.11n 模式的 WiFi 无线网卡，其理论速度可以达到 300Mb/s。有很多疯狂 WiFi 的 DIY 玩家，制作高增益天线以及高灵敏度发射机，将 WiFi 信号发射到几十公里外而仍然保持良好的信号质量。由此也就没有理由怀疑，在不久的将来，以太网 HUB 及交换机也将被淘汰。新建的楼房不需要将昂贵的铜制双绞线埋入墙中，而只需要在走廊上安装一个 WiFi 信号发射/接收机即可。

然而，人们总是有一种复古的倾向，新思想替代旧思想终归需要一段时间。目前的新建筑仍然要埋置双绞线到墙内。

附图 2 所示的凌乱布线，在无线时代，将被认为是地球上最不可思议的照片。就像现代人看原始人钻木取火一样。



附图2　凌乱的布线

随着人类对世界规则的不断探索，或许可以发现像量子一样更多新奇的东西，在这些知识的基础上，可以发明出更多的新奇技术。世界将会不断的改变它的模样。

本书就此告一段落。再见！



后　　记

随感

殚心竭力两周年，
谁解个中苦与甜？
古今多少兴衰事，
成败从来在眼前。
——敖青云

各位朋友，非常感谢您能看完此书。如果您对这本书有何建议和意见。可以发送邮件到 myprotein@sina.com，我当万分感谢！

另外，还可以到本人博客留言或者邮件讨论本书相关的内容。

最后，实在想不到拿什么送给各位以表谢意，就送各位一首诗，也送给我自己。

书山有路勤为径，
学海无涯苦作舟。
主机网络和存储，
做得 IT 皆英雄！

书中角色最后归宿：

七星大侠：开天鼻祖，光芒永照。

张真人：百年求道，一生孜孜不倦、德高望重、悬壶济世、鞠躬尽瘁、死而后已。

微软老道：承蒙张真人赏识，不负众望，成为武林盟主。

无忌：革命之后，到处求仙访道，不知其踪。

老 T：把持武林交通系统，依然向最后一块阵地不断进攻。

FC 大侠：把持着那最后一片领土，与老 T 对峙到底。

冬瓜头
大连七贤岭